MIT/LCS/TR-323

EFFICIENT IMPLEMENTATION OF APPLICATIVE LANGUAGES

William Beekley Ackerman

Efficient Implementation of Applicative Languages

by

William Beckley Ackerman

S. B., Massachusetts Institute of Technology
1967

S. M., Massachusetts Institute of Technology
1977

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

March, 1984

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
March 28, 1984

Certified by ______________________________
Jack B. Dennis
Thesis Supervisor

Accepted by ______________________________
Arthur C. Smith
Chairman, Departmental Graduate Committee

Efficient Implementation of Applicative Languages

by

William Beckley Ackerman

Submitted to the Department of Electrical Engineering and Computer Science
on April 4, 1984 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

The analysis of parallelism in an applicative program is much easier than in a program written in a conventional statement-oriented style. This makes it possible for an optimizing compiler to prepare such a program for extremely efficient execution on a suitable enormously parallel computer. This thesis explores the transformations that must be made to achieve very high performance for numerical programs when executed on a computer that uses data flow principles in its operation.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Computer Science and Engineering

Keywords: applicative programming, parallel computation, optimizing compilers, data flow computation.

ACKNOWLEDGMENTS

I would like to thank my thesis supervisor, Jack Dennis, for his encouragement and support during the course of this research, and for providing the intellectually exciting environment that is the Computation Structures Group.

My other thesis readers, Bert Halstead and Arvind, have been very helpful in this work, and I have enjoyed many wide-ranging discussions with them on many topics in the field of computer science.

My many friends in the Computation Structures Group have been a continual source of stimulation, ideas, and companionship. I would particularly like to thank Nena Bauman, Andy Boughton, Dean Brock, Gao Guang-Rong, Clement Leung, and Willie Lim for their contributions to this work and their friendship over the long period that it has been my pleasure to know them.

This research was supported by the Department of Energy under grant number DE-AC02-79ER10473, and by the National Science Foundation under grant number MCS-7915255.

Table of Contents

1. INTRODUCTION

There is an enormous literature of numerical algorithms, for such applications as hydrodynamics, image processing, and weather forecasting, that require very high performance computers, much higher than can be attained without the use of parallelism. These algorithms are commonly executed on vector, pipeline, or array "supercomputers", which exploit parallelism in the program to a limited extent. These algorithms can in fact exhibit incredibly more parallelism than conventional supercomputers exploit. (This is especially true since, as faster computers become available, these programs are written to use an ever finer grain in their analysis or simulation of physical phenomena, which increases the parallelism even further.) The reason that conventional supercomputers fail to fully exploit the parallelism is that they can only take advantage of local parallelism. Their design has a "bottleneck" between the control section and the arithmetic processing section. This bottleneck makes it impossible to exploit parallelism except in a local and restricted way. Tremendous overall speed can be achieved if programs are executed on a computer designed to eliminate the bottleneck and take advantage of parallelism in a more general way. Applicative programming makes such computing systems feasible. The goal of this thesis is to show how to break the bottleneck and exploit the parallelism.

The bulk of this thesis is a presentation of transformations that can be made to applicative programs to achieve high performance. These transformations should be made by an optimizing compiler. This thesis can therefore be thought of as a description of the principles on which such an optimizing compiler ought to be based.

There is really only one aspect of computation that makes the program translation problem difficult -- arrays and other "aggregate" structures such as records. For this reason the major focus of the thesis will be array computation. It will be shown that, if the program is written in an applicative language and represented in the form of a data flow graph, the potential parallelism can be exploited in a natural way. A suitable direct-execution computer for such graphs, that is, a *data flow computer*, should be able to achieve extremely high performance in executing such a program.

1.1 Background

Array processors like the Illiac IV [12] have many processors operating in lock-step. This makes it possible to do repetitive array operations at a speed that is not limited by one's ability to shave nanoseconds on one processor. This in turn makes the bottleneck extremely fast (potentially as fast as we like, for sufficiently large arrays and sufficiently large computers), but it is still a bottleneck. The reason is that the computer is still doing essentially only one thing at a time. Vector and pipeline processors [28] perform repetitive operations on arrays sequentially, but with substantial overlap from one element to the next.

In vector, pipeline, or array processors, the only places where massive speedup can be achieved are repetitive, nearly identical computations on arrays where there is no data dependence among the computations on the various array elements. That is, the only data that are available for the computation of a particular array element are the data in corresponding positions (possibly with shifted indices) in other arrays (including "mask" arrays of control information that can enable or disable certain operations) and global data that affect the computations of all elements.

This amounts to an inflexibility that seriously limits the parallelism exploitable by such machines, in comparison with the parallelism achievable in principle. Among the lost opportunities for parallelism are:

1. Scalar operations. Although high performance computers sometimes have impressive mechanisms to speed up scalar computations through data caches, instruction caches, instruction lookahead, and "on the fly" optimum scheduling of functional units, the vector, pipeline, or array hardware of the big supercomputers is completely ineffective for scalar operations. As supercomputers become more powerful in their ability to process regular arrays, this "scalar bottleneck" becomes proportionally more serious.

Consider the following fragment of a program to be executed on a conventional supercomputer:

A a very intense vector computation

B some scalar computations that do not depend on the outcome of A, but are needed before C can begin

C another very intense vector computation

Assume all three blocks take about the same amount of real time, but of course blocks A and C contain vastly more computation than B and use the computer's vector mechanism to perform it. Block B could in principle be executed simultaneously with A. Since it actually performs an insignificant amount of computation in comparison to A, it would be very little additional burden to the computer's processing units. It would double the computation speed (assuming a long sequence of these alternating vector and scalar blocks) by letting the intensive computation C commence as soon as A completed, thereby keeping the vector mechanism busy all of the time rather than half of the time. The problem here is that, while the arithmetic units are very powerful, the computer's control mechanism lacks the flexibility to schedule operations in this manner.

2. Irregular array operations. Conventional supercomputers have very little ability to perform different computations on different elements of an array. Their ability is generally limited to use of "mask" arrays that can enable or disable specific instructions on specific array elements. This, too, is a lack of flexibility in the scheduler. The computer's vector mechanism runs at such a high speed that it doesn't know what it is doing in terms of the control structure of the program. It does not have time to stop and think about individual array elements.

3. Sequential data dependencies within iterations. Some operations on arrays have an actual data dependency from the computation involving one element to the computation involving the next. True simultaneous processing is of course impossible here. However, a large amount of parallel processing is usually still possible, because only a small fraction of

the arithmetic operations are in the "critical path". Conventional supercomputers, however, can't take advantage of this because the operations in the critical path are deep inside the pipeline and can't be "fed back" at full speed from one point in the pipeline to an earlier point.

All of these shortcomings arise from one fundamental problem: the control mechanism ("scheduler") of conventional supercomputers lacks the flexibility to take advantage of the parallelism actually present in the algorithm.

1.2 Data Flow Analysis with Scalars

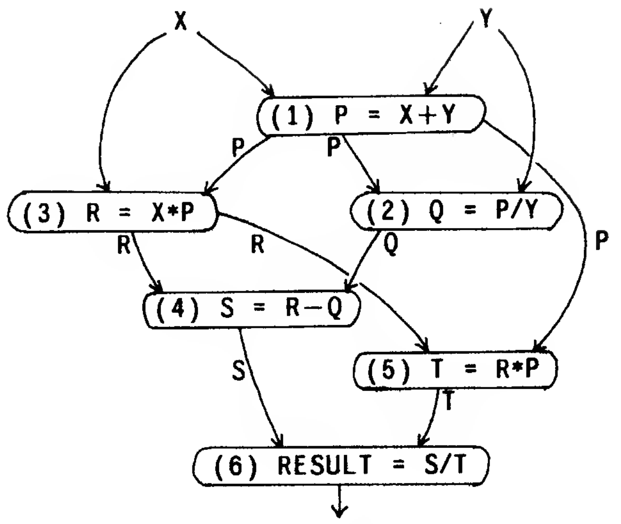
For algorithms not involving arrays, the method for "optimally" scheduling operations to take full advantage of parallelism is well known. It is the data flow model. The instructions of the program are arranged in a "data flow graph", showing their interdependencies.

For example, the FORTRAN program

```
1        P = X+Y
2        Q = P/Y
3        R = X*P
4        S = R-Q
5        T = R*P
6        RESULT = S/T
```

would have the following data flow graph:

Fig. 1.1

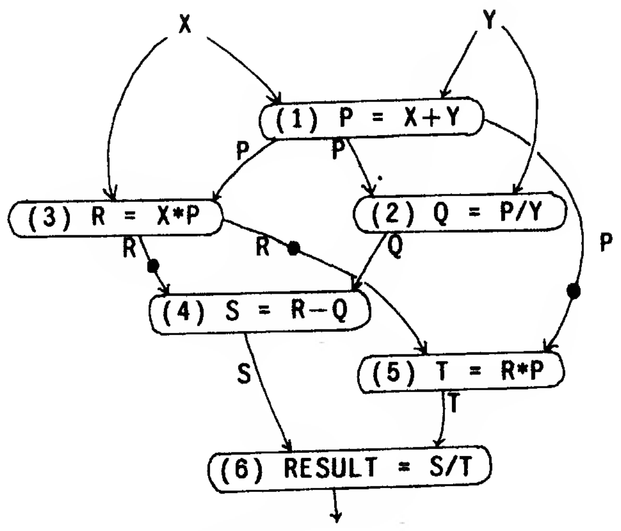


Data items are transmitted in programs in FORTRAN and most other languages through the writing and reading of variables. In a data flow graph this transmission is shown by an arrow from the source operation to the destination operation. For programs that deal only with scalars, the translation from conventional languages is not too difficult [7, 20, 46].

The ideal parallel scheduler will execute each operation as soon as the operands that it requires become available. One can visualize the operation of such a scheduler through the "token" model: Results of operations are created in the form of tokens (shown as black dots) that flow along the arrows. The ideal scheduler will therefore execute each operation when it has tokens present on all of its inputs.

A possible state of the execution of the above graph might be

Fig. 1.2



1 & 3 have completed
2 is undergoing execution

Any scheduler that uses this criterion for instruction execution will find all opportunities for parallelism, without regard for locality. The possible parallel instruction executions will not be limited to a single locus of control, or even a local "window" of control, or even some collection of regular, repetitive operations. Computers capable of direct execution of programs encoded as data flow graphs have been proposed [9, 16, 17, 22, 21, 33, 40], and some prototypes have been built.

The design of practical direct execution data flow computers requires a few enhancements and refinements to this model. Some of these refinements will be described briefly here -- detailed consideration is not crucial to this thesis. A *dynamic* data flow graph can effectively expand and contract during execution. This is typically done when a function is invoked. A function invocation is initially encoded as an "apply" node. When a token reaches that node, it turns into (that is, the computer operates as though it had turned into) the graph representing the function body. A *static* data flow computer can only handle static graphs. In static graphs, function invocations are expanded at compile time, before execution begins. This necessarily precludes general recursion at the hardware level.

Another refinement is the use of special nodes to control conditionals and iterations. In the standard model data flow graph [23, 24] these nodes are called "T", "F", and "MERGE". The "T" gate takes one argument of arbitrary type and one boolean argument. If the second argument is *true*, the first argument is passed on as the result. Otherwise it is simply discarded. The "F" gate does the opposite. "T" and "F" gates control the initiation of conditional computations and the recycling of values during iterations. The "MERGE" gate takes two arguments (labeled T and F) of arbitrary type and a third argument which is boolean. It accepts and passes on as the result whichever of the first two arguments is indicated by the third argument. The unused argument is not absorbed and is not even required for the operator to fire. This operator is used to merge the results of the various arms of a conditional, and to define the initial loop variables for an iteration. See [18, 46] for a detailed description of the use of these gates. In this thesis, data flow graphs will be shown in a level of detail that does not involve these gates -- conditionals and iterations will be shown in terms of the more abstract nodes described in Section 2.3.

1.3 Applicative Programming

For algorithms that deal with arrays, achieving "maximally parallel" execution is a far less tractable task. One reason for this is fundamental and will be discussed in Section 1.4; the other is that array computation is usually not performed applicatively. This second reason is not fundamental -- it is an artificial problem arising from conventional ways of conceptualizing computation. It must be disposed of before the real issues can be addressed.

The processing of data in arrays is commonly perceived in terms of the "von Neumann" architecture, which is not compatible with true data flow analysis. In this model of architecture, one conceives of a computer as a machine that manipulates data stored in a memory, by executing commands in some sequence. For programs that deal with scalars, it is not difficult to recast one's thinking in terms of data dependencies instead of sequential command execution, nor is it difficult to translate programs into data flow graphs. The reason is that the "memory slots" of the machine's memory can be thought of as nothing but "pipes" between arithmetic operations, that is, variables in the program become arcs in the data flow graph. Hence, the von Neumann style of thinking is not a serious burden in this case

The von Neumann style of thinking has a much more profound effect when arrays enter the picture. Instead of being simply "memory slots", the memory consists of "blocks" that are treated as having a very real existence. The need to pass entire arrays from one part of the program to another, and the expense of copying them, means that the memory blocks that arrays occupy have a very real existence in one's conception of what the program is doing, and in the execution model that a programming language presents to the user. Execution of high level languages on von Neumann computers is customarily performed in a way that requires that this execution model be pervasive in the external behavior of the "virtual machine" that defines the language. So pervasive, in fact, that for a great many high level languages, such as FORTRAN, BLISS, and C, the target machine for which the language was designed is clearly discernible from reading the language manual.

This intertwining of the von Neumann conception of the way arrays are manipulated with the way humans write programs makes the bottleneck difficult to break. Vector and pipeline supercomputers are designed to do their vector operations by regular manipulation of successive words in blocks of memory, because that is the way programs are conventionally written. It is nearly impossible to get truly flexible parallel execution without getting away from the notion of memory blocks manipulated exactly the way the programmer specified. But it is nearly impossible for a compiler to translate a program as anything other than such a memory block manipulator, because the actual data dependencies in conventional programs are so obscure.

What is needed, then, is a style of programming that does not enslave the computer to the von Neumann style of execution, and a style of translation and execution that can truly exploit the parallelism. Such a style of programming exists: it is called applicative programming [4]. In applicative programs, all data dependencies, even those involving arrays, are apparent.[1] (The reason it is easy to deduce the data dependencies of programs that do not use arrays is that such programs are essentially applicative anyway, no matter what language they are written in.)

For the purposes of this thesis, the important characteristic of applicative languages is the following: Whenever any array (or other data aggregate such as a record) is modified by some computational step (say A), the effect of that modification must always be made known, by some mechanism, to other computational steps (say B). If the transmission of that effect takes place because B "knows" where the array is, perhaps having known its location since before A modified it, the language's model is not applicative. Most languages are not applicative -- in most languages those statements that refer to the array X are compiled into code that knows where X is allocated, whether on the stack, in a COMMON block, or in some local or global memory space, and the code manipulates X through that predetermined address.

This type of transmission of effect (called a "side effect") is not compatible with data flow scheduling because the constraint that step A must precede step B does not take the form of a token transmission from A to B.

In an applicative language model the only way computational step A affects step B is through direct transmission of its result value to B. The result of A is a new array (conceptually at least), containing the outcome of A's action. Only steps that receive this result can sense A's action -- no other step can sense it indirectly by reading the array that it knew A would modify. (Of course, in an implementation, step A may very well write into memory. It is the job of the implementation to make sure that the behavior is consistent with the applicative model.)

1. We do not consider data dependency removal that might be effected by rearranging the algorithm, or by using mathematical identities beyond the normal purview of a compiler.

This type of transmission is fully compatible with data flow scheduling. One allows tokens to carry array values as well as scalar values. Array operations are performed by functional operators such as the ones to be described shortly.

The use of applicative arrays is a very radical departure from the conventional methods of computation in numerical applications. It has led to widespread and severe criticism regarding its efficiency in such applications. A major goal of this thesis is to answer these criticisms.

A few words about array nomenclature are in order. The differences between applicative arrays and the "arrays" (or "subscripted variables") of conventional languages are profound. It is very easy to use slippery language that confuses the two concepts. In an applicative system, an array is nothing more or less than a series of values, along with the low and high bounds information indicating between what index limits these values lie. The array *is* the information. In von Neumann style thinking, on the other hand, an array is a *place* where values may be stored in sequence.

Strictly speaking, it is therefore incorrect to speak of "writing X into array A at position J." One should only speak of "producing an array with X at position J and otherwise identical to A." In practice, however, the way array computation is performed in numerical programs is not much different in a von Neumann system or an applicative one. In FORTRAN, an array is filled by subscripted assignments to it, typically in DO loops. One can quite properly speak of "writing X into A at position J." In an applicative language, one would use an iteration variable which is an array and which is repeatedly rebound. One should speak of "rebinding the iteration variable A to an array containing X at position J and otherwise identical to A's old value." In practice, the two operations are analogous to each other and are used in the same way in algorithms written in conventional and applicative systems, respectively.

In this thesis, the more conventional nomenclature will often be used, even though applicative array operations are being described. The reader should be aware that "writing into an array" is something of an abuse of language in this context.

1.4 Array Computation in Applicative Systems

Having disposed of the fictitious problems with array computation, we can consider briefly, if imprecisely, what it is that really makes array computation difficult to deal with.

The fundamental characteristic of arrays that makes computation with them difficult to analyze is that arrays, in general, contain an amount of data that is incommensurate with the size of the program structures that manipulate them. Programs that manipulate scalars only need to deal with as many data items as there are names to denote them appearing in the text of the program. (Of course, it can be argued that the information content of a small collection of scalars can be increased without limit by increasing the precision, that is, the word length. However, this has no effect on the complexity of the program's behavior as long as the data are treated as indivisible units. If the program looks inside the data words instead of treating them as indivisible units, they become, in effect, arrays.)

Any arrays that are of small fixed size present no problem.[1] The reason is that such an array can be treated simply as that many scalars. Data dependencies can be analyzed for the various array elements individually. Random accesses ("subscripted array references") can be rewritten as conditional program structures that test the index value one case at a time.[2]

1. The meaning of "small" is of course subjective. Any number is small if we are willing to make the control structures large enough. Practical experience tells us, however, that 3 is small and 100 is large.
2. It is an aversion to 100-armed conditionals that leads us to say that an array of 100 elements is not "small". It is clearly not efficient to let the control structure of a program grow in proportion to the size of the arrays.

It is when arrays are large, however, or when their sizes are unknown in advance and hence might conceivably be large, that they exhibit their difficult behavior. In this case, program structures such as conditionals can no longer encompass the range of the possible array indices. Then the indexing operations for reading and writing arrays become essential and nontrivial. When a compiler sees just an expression like

```
A[J]
```

it knows nothing, in the most general case, except that J is the result of some integer computation which it perhaps can't analyze, and which it must use to fetch the appropriate element from the array A, whose internal structure it can't discern.

The fact that the manner in which data are written to and read from arrays is often unpredictable makes it very difficult to organize programs for extremely efficient parallel computation. When unpredictable, or "random" access to arrays occurs, it is desirable to store the array in a localized area in physical memory, so that the hardware will be able to make the accesses readily. This is the antithesis of truly bottleneck-free parallel distributed computation.

Arrays that are accessed unpredictably are the bane of conventional supercomputers also. The vector and array mechanisms of these machines can only deal with fairly regular algorithm structures. The various optimizing compilers and program transformers for these systems search the source program for computational structures (DO loops, typically) that can take advantage of the machine's array mechanism.

1.5 Spatial Distribution of the Computation

The way to achieve very high performance is to distribute the computation across many distinct processing units. Since we envision execution on a data flow computer, the first step is to distribute the computation across a data flow graph. The second step is to map that graph onto many processing units in such a way that those units can proceed as nearly independently as possible and with as few constraints as possible. The mapping problem, while extremely important, will not be addressed in this thesis. We will consider parallelism that is visible in the data flow graph to be parallelism achievable by the computer. In effect, we will consider each instruction to have its own processor, so that "ideal" execution is obtained -- each instruction in the graph can fire when its operands are available, independently of anything else that is happening elsewhere in the computer. This allows us to ignore the mapping problem. In an actual computer, one processing unit would have to serve the needs of many instructions, so the mapping must be chosen to balance the loads on the processors.

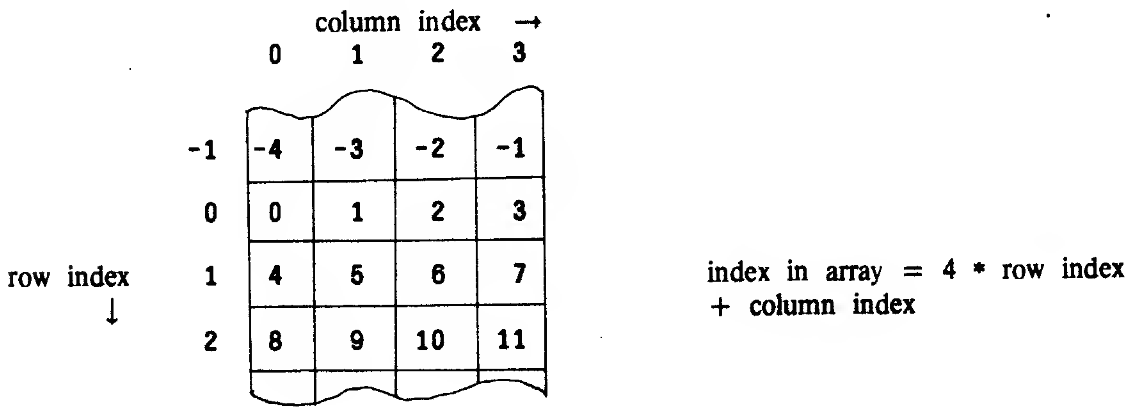
The representation of a program as a graph constitutes a distribution of the program over many processors and consequent exposure of its parallelism. This amount of parallelism is not enough. Additional parallelism, enormous amounts of it, will be found among the different cycles of iterations ("loops"). To expose that parallelism in the data flow graph, the cycles will be *unfolded* (or "expanded" or "unrolled"). This is the principal technique whereby huge increases in computation speed are obtained. Of course, this unfolding takes up space in the machine representation of the graph. Very high performance computers will have to have a large "instruction space". The total speed of a system is limited by the product of the instruction space and the processing speed per instruction. (The latter quantity depends on the speed of each processing unit and the number of instruction cells that it must serve, that is, the degree of deviation from ideal execution.) This thesis will be oriented toward extremely large machines executing programs whose loops have been unfolded accordingly.

1.6 Spatial Distribution of Arrays

The translation of programs written in an applicative source language into static data flow graphs is not too difficult a task. This is so even if array operations are used, if the execution mechanism for the graphs supports the array operations directly, exactly as they appear in the source program. Such a direct hardware support for arrays means that arrays appear as tokens in the data flow graph, just as scalars do. The operations for reading and writing arrays look like any other operations in the graph -- argument tokens flow into them and results flow out. Implementing such a scheme is rather tricky. Since it is not feasible to carry all of the array data in an actual token, a method must be worked out whereby the array data are stored in a memory unit and some sort of pointer to it is carried in the token. It is the job of the implementation (hardware, firmware, and compiler) to ensure that the resulting mechanism is equivalent to the idealized mechanism in which the entire array is carried in the token. The details of such a mechanism are beyond the scope of the thesis. The interested reader is referred to [3], in which an implementation is described for general dynamic arrays, or [25] in which it is described for a restricted class of array operations for which efficient execution may be obtained very cheaply. For the purposes of this thesis, it is assumed that some appropriate hardware mechanism exists.

This still leaves a serious bottleneck in the handling of arrays, especially in unfolded loops operating on a very large, high performance computer. To circumvent this, as we expand the loops, we will expand the arrays also. The general representation and translation technique that will be used is *spatial interleaving*. A linear array is rearranged as if it were a two-dimensional strip of fixed width. If the given array's size is known at compilation time, the strip's length is known as well.

Fig. 1.3



Each column is then treated in the hardware as a true array, which is implemented by whatever mechanism the hardware provides. These column arrays are handled independently. High speed is achieved, as on conventional computers, by wide interleaving, that is, by making the strip very wide and having many independently handled arrays. Unlike conventional computers, the separate arrays do not need to pass through the same bottleneck, so the amount of possible interleaving is virtually unlimited. The interleaving is performed by the compiler and is invisible to the programmer. It is the job of the compiler and the hardware to make the overall computation faithful to the programmer's (uninterleaved) intentions.

When the array is more than one-dimensional, a similar interleaving is used in each dimension.

1.7 Regularity of Array References

The techniques to be shown in this thesis work well only for array operations that take place in regular, repetitive patterns. The bulk of the array references (reading or writing) must be inside iteration loops, with the array index of the datum being accessed related to the cycle number in the loop by an *affine mapping function*. An affine function is a first degree polynomial, that is, a function of the form $f(x)=Ax+B$. A typical iteration that obeys this rule is something like

```
          DO 10 I = 1, 1000, 2
          .......
          ...  A(4*I+35) ...
          .......
      10  CONTINUE
```

We can actually relax the restrictions in a number of ways at modest cost in efficiency. These relaxations will be presented later in the thesis. One relaxation is that the function mapping the relative time of data access to its array index can be a *piecewise-affine* function. This means that it can be broken up into a "small" number of pieces, each of which is of the form $f(x)=Ax+B$. The question of how small is "small" is answered in the usual way: it should be small enough that program structures of that size (e.g. a conditional with one arm per piece of the array) are feasible. The number of pieces is typically very much smaller than the size of the array itself.

Common examples of piecewise-affine array mapping can be found in the handling of boundary conditions. All elements of an array except the first and the last are created in a regular sequence by a uniform method, but the first and last elements are computed as special cases. Such an array might be expected to be produced in three pieces.

Perusal of common numerical application programs shows that they generally satisfy the restrictions on regularity of array access. Of course there are exceptions, but those exceptions are relatively rare and do not usually occur in the computationally intensive part of the programs. It follows that, for many numerical calculations, tremendous performance improvement can be obtained through the use of the data flow principles to be presented.

1.8 Synopsis

Chapter 2 will define the conventions and notation to be used in the rest of this thesis. Chapter 3 will describe the two basic operations that we will perform on programs to obtain high performance: *unfolding* of loops and *interlace* of arrays. Chapters 4 and 5 will describe how those operations will be made to work in practical situations. Chapters 6 and 7 will describe additional important transformations. Chapter 8 will discuss the special problems that arise when programs "write" into arrays. Chapter 9 will discuss the types of transformations that must be made when the compiler has less information about array reference indices than we would like. Chapter 10 will describe how parts of a program can be permitted to get "out of step" with each other for improved performance. Chapter 11 will describe a final transformation of arrays just before they are mapped onto hardware structures. Chapter 12 will describe how occasional "irregular" array references are handled. Chapter 13 will show how these methods apply to programs with nontrivial structure, presenting two real example programs: a tridiagonal equation solver and the fast Fourier transform. Chapter 14 will describe some special array optimizations that can sometimes be made. Chapter 15 will present a few thoughts about the design of the actual computer upon which these transformed programs should run efficiently. Chapter 16 will present some conclusions about the manner in which a practical optimizer might operate.

2. THE EXECUTION MODEL AND ITS NOTATION

The execution model that will be used is the static data flow graph. This is an ideal model for the study of parallelism in applicative programs, since the data flow graph of such a program faithfully represents the possible parallelism in its execution. Since there are no side effects, the execution of any operation can take place as soon as its immediate predecessors in the graph have computed their results. It does not need to wait for any other control signal; no other activity in the program could possibly affect it.

The appropriate computer for this model is an idealized "data flow computer" that can "fire" (execute) all operations independently. Such a computer, in the ideal limit of true simultaneous execution of *all* enabled operations, can take ultimate advantage of the parallelism in an applicative program.

In the static data flow model, the data flow graph does not change its structure during program execution. This property assumes that functions are expanded "in line", prior to execution, as if by a macro preprocessor, and makes execution of recursive functions very difficult. The graph consists of nodes which can perform predetermined applicative operations, and directed arcs carrying data "tokens" among the nodes. Except for the special nodes to control conditional and iterative computation, the nodes have a straightforward firing rule: a node can fire whenever its input arcs contain tokens and its output arcs do not. When it fires, it absorbs its input tokens, using the data they contain as arguments to compute its function, and places a copy of the result of that computation on each output arc to be sent to other nodes.

As noted previously, the translation of an applicative program into a data flow graph is not difficult if only scalar values are involved. Likewise, the qualitative analysis of how such a graph would be executed on any reasonable data flow computer is trivial. When arrays are involved, the transformations that must be made become rather involved.

Since all array operations must be performed as applicative operations encoded in the data flow graph, a set of "basic" machine level operations suitable for high speed computation must be chosen. These are the data flow computer's analogue of the indexed load and store operations that are the basic array instructions of conventional computers. The fundamental operations are

```
select      written as: A[J]
append      written as: A[J:V]
```

Select performs the same "subscripted read" operation as in conventional languages. **Append** returns an array similar to the argument "A", but with value "V" at index "J", replacing whatever "A" has at that index. While different operations can be devised, applicative programming requires that any "subscripted write" operation be somewhat similar to the **append** operator.

There are a number of alternative forms of the **append** operator. Two that are especially useful for the sequential creation of arrays are:

```
addh        written as: addh(A,V)
addl        written as: addl(A,V)
```

These return an array whose upper bound has been increased, or lower bound decreased, respectively, with the given value as the new element. (We assume that the low and high bounds of an array are part of the information associated with the array.)

2.1 The Source Language

Programs will be written in a suitable subset of VAL [5]. This language is, for the purposes of this thesis, a rather ordinary and typical language. It is applicative, it has the desired array operations, and it has suitable control structures. The properties of VAL that set it apart from other languages such as Id [9] or pure LISP [42] relate mostly to syntax and type checking, which are irrelevant to our concerns here. VAL is rather simple-minded in its approach to functions, in that functions are not treated as "first class objects".[1] This makes it suitable for translation into static data flow graphs. Languages that treat functions more generally, such as Id, LISP, FP [11], or KRC [45], may be more difficult to translate for the execution efficiency that we envision for VAL. (This is not to say that they will never be implemented as efficiently as VAL, but we believe that further developments in architecture and compiler design will be required.) The restrictions imposed by using VAL and static data flow graphs do not impose any undue hardship for many application areas.

The principal array operators of VAL are `SELECT` and `APPEND`, described previously. When building arrays with **append**, one often starts with the array "`array_empty`". Because VAL is applicative, this is a *constant.* The semantics of VAL provides detailed rules governing the behavior of `SELECT`, `APPEND`, and `array_empty`, particularly regarding how array bounds "stretch" when an `APPEND` writes into a position out of the existing bounds. These rules will be ignored in this thesis -- they are not important for the kinds of regular array creation we will be dealing with. We will just assume that application of a series of `APPEND` operations to `array_empty` will always do the right thing. We will not even assume that array bounds are part of the data carried by the token. The difference between the assumptions being made and the actual semantics of VAL (or any similar language), and the difference between those assumptions and the actual behavior of a target data flow computer, are details that may be left to the design of a compiler.

Other array operations have been proposed for the VAL language, but they are not important. Most of them perform "housekeeping" operations involving the array bounds.

1. This means that function names and definitions may not be passed as parameters, bound to variables or arrays, or otherwise manipulated. The only thing that one may do with a function is invoke it, and the only thing that one may invoke is a function name which is permanently bound at compile time to a function definition that the compiler can find.

The notation for iterations in VAL is shown in the following example:

```
for J, K := N, 1                % J and K are loop variables
do  if J = 0 then K             % exit and return this value
    else iter J, K := J-1, J*K enditer
    endif
endfor
```

The body of a **for** block may evaluate (depending on the values of test expressions involving the loop variables) to an **iter** clause. If so, the loop variables are bound to the new values as indicated, and the body is evaluated again. Otherwise, evaluation of the **for** block is complete. As noted in [4], this is typical of the way iterations must be expressed in an applicative language.

The VAL language also has a **forall** block exemplified by

```
forall J in [LO, HI]            % "vector sum" of A and B
construct A[J]+B[J]
endall
```

In this thesis, the **forall** block will be considered as nothing but a "sugaring" (short notation) for the appropriate **for** block. All considerations of iterative program structures and their transformations will be made in terms of **for** blocks, and the extension to **forall** blocks will be implicit. For example, the above **forall** block can be considered a sugaring for[1]

```
for J, A := LO, array_empty
do  if J > HI then A
    else iter J, A := J+1, A[J: A[J]+B[J]] enditer
    endif
endfor
```

1. This ignores a minor difference in the way the VAL semantics treats pathological values of the array bounds.

One may wonder why we artificially sequentialize the **forall** block. The object of the game is to uncover parallelism, and the **forall** block is designed to show the compiler where parallelism lies, so destruction of this information seems counterproductive. The reason we do this is that the correct translation of the **forall** block is the same as the correct translation of the equivalent **for** block, as will become clear later when "loop unfolding" is introduced. When a **forall** block is converted into an iteration and a loop unfolding of `N` is performed, the resulting code simultaneously computes `N` consecutive instances of the **forall**, and then moves on to the next `N`, and so on. If `N` is as large as the machine size will support, this is the best way to translate general **forall**'s.

If the **forall** limits are known to the compiler and are reasonably small, converting it to an iteration loop is still the correct thing to do. Complete unfolding of the resulting iteration will generate the "intuitively obvious" parallel **forall** code.

Whether an optimizing compiler actually converts **forall**'s into iterations, or just acts as though it had, is a minor implementation detail.

2.2 Compile-time Parameters

It will soon become apparent that it is very important for an optimizing compiler to know the numerical values of certain parameters such as array sizes. Of course it is not appropriate for such parameters to be specified as manifest constants at the points where they are used. The Fourier transform procedure, for example, should be written with the array size as a symbolic parameter:

```
function DFT(ARG: array[complex] ; N: integer . . .
```

(The reader who is curious about what comes next should refer to Section 13.2.)

In any given run of a program, it will be known, for each invocation of this function and others like it, what the value of **N** will be. Typically, the programmer will want to specify the numerical value at the outermost level of the program, leaving it as a symbolic value everywhere else. It is for this reason that some languages such as FORTRAN77 [29] and ADA [6] have a "parameter" or "constant declaration" feature. Using "parameters", values that are required to be known prior to execution can be specified numerically only at the outermost level, and left in symbolic form elsewhere.

A nice property of functional languages is that there need be *no difference* between such a "parameter" and an ordinary "variable". There is no need to tell the compiler that, in one case, a parameter will be numerically specified at the start of the program, and in the other case, that a variable should have memory allocated to it. Since a "variable" in a functional language can't be changed once it is defined, there is no need to allocate memory for it.

A definition such as

```
let N := 2048
in  . . .
    . . .
    . . .
endlet
```

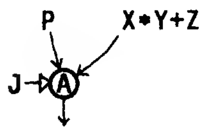
is completely equivalent to substituting **2048** for all free occurrences of **N** in the block. The same is true for definitions made by passing arguments to functions.

Therefore, it is reasonable that "compile-time parameters" will be specified, using normal mechanisms of the language, by a top-level function such as the following:

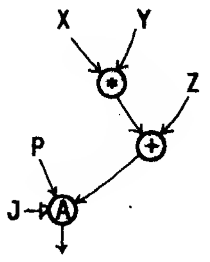
```
function TOP_LEVEL(DATA: array[real] returns array[real])
    SOLVE_NAVIER_STOKES_EQUATION(DATA, 2048)
endfun
```

2.3 The Graph Model

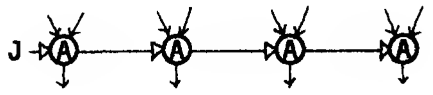
In this thesis data flow graph fragments will be exhibited in a very simplified notation designed to convey just the essential aspects of each graph. Ordinary connections among operators will be shown in the natural way, with arrows. The select and **append** nodes will be illustrated by nodes with abbreviated names S and A. The *index* argument to these nodes will be shown with an open arrow: —▷Ⓢ . Where possible, that arrow will be horizontal. The other argument(s) will be shown with plain arrows. The **append** operator takes two other arguments. The one on the left will be the incoming array; the new value will be on the right. Where the actual graph structure would be more confusing than helpful, ordinary expressions will be written.



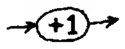
means



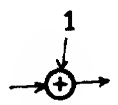
Where several nodes receive the same index, an arrow will pass through all of them:

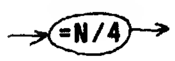


Arguments will occasionally be subsumed into nodes:

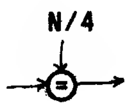


means



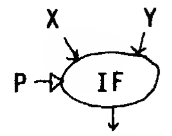


means



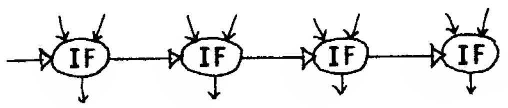
The ubiquitous constant `array_empty` will be written as "Λ".

The notation for conditionals is as follows: An IF node takes a boolean argument on the side, and data at the top. The left data argument is used if the boolean is true; otherwise the right argument is used.

`if P then X else Y endif` is represented as 

The actual representation of a conditional in a static data flow computer is quite different -- a "MERGE" operator plays the part of the IF, but "T" and "F" gates are placed at the top of the subgraphs computing X and Y to allow tokens to enter only the chosen subgraph.

Several IF's in a row can have a single boolean arrow going through them:



Iterations have the most complex notation. (This includes loops made from the VAL **forall** feature -- the latter is treated as an abbreviation for a sequential loop.) The body of the loop is surrounded by a dotted line passing through "LV" nodes at the top and a FOR node at the bottom. The "LV" nodes are sources of the Loop Variables for each cycle. The data shown going into them at the top (outside of the dotted line) are the initial values, computed only once. Just above the final FOR node is an ITERIF node controlling the loop. It is a variation of the IF node of ordinary conditionals.

The body of a VAL iteration loop is an if/then/else, one clause of which is of the form "iter <vars> := <newvalues> enditer", and the other of which defines a tuple of ordinary values.[1] That if/then/else will be an ITERIF node. The boolean control value will enter on the side in the usual way. The "data values" going into it will be "iterating arms" or "returning arms", drawn as boxes with "I" or "R" in them. An iterating arm has as many arrows going into it as there are loop variables. If that arm is selected,

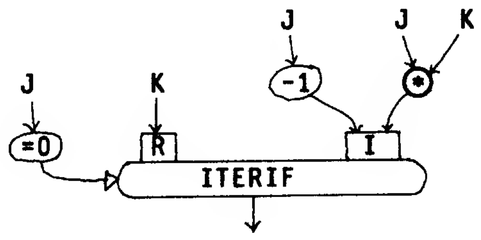
1. The VAL language actually allows many variations of this.

those values are used as the new loop variables, and another iteration cycle takes place. That is, the tokens are considered to be routed back to the respective "LV" nodes and to re-enter the loop body. A returning arm has as many arrows going into it as there are values returned by the entire loop, which is the number of arrows emanating from the FOR node. If a returning arm is selected, those values are returned. That is, the tokens are considered to pass through the FOR node and out of the loop.

If J and K are the loop variables, then

```
    if J = 0 then K
    else iter J, K := J-1, J*K enditer
    endif
```

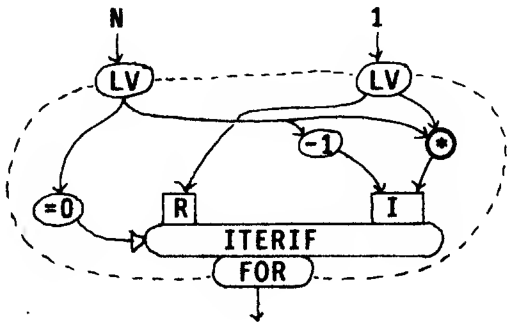
is represented by



and

```
    for J, K := N, 1
    do  if J = 0 then K
        else iter J, K := J—1, J*K enditer
        endif
    endfor
```

is represented by



3. ITERATION UNFOLDING AND ARRAY INTERLACE

The principal program restructuring operation that we will perform is that of *loop unfolding*, also called *loop expansion*. This basically consists of writing out multiple copies of the loop body. In a data flow or other suitable parallel computer, these copies are expected to execute simultaneously, instead of the forced sequential execution implied by the original iteration.

Recall that, in an applicative language, the purpose of an iteration is to apply some transformation to a set of loop variables repeatedly, until some predicate is satisfied. We will be primarily interested in transformations that are performed a certain number of times, known in advance. These are characterized by having one of the loop variables be a counter, and having the predicate be a test of that counter and the transformation involve incrementing it.

The standard iteration might look like

```
%% compute f^N(INITVAL)
Z := for X, I := INITVAL, 0
do  if I=N then X
    else iter X, I := f(X), I+1 enditer
    endif
endfor
```

In a static data flow computer, and most other forms of computer, this suggests only one locus where f is computed, and hence strictly sequential evaluations of f, unless the situation can be improved upon.

A simple form of unfolding would be to rewrite this as

```
%% compute f^N(INITVAL), N is even
Z := for X, I := INITVAL, 0
do  if I=N then X
    else iter X, I := f(f(X)), I+2 enditer
    endif
endfor
```

This transformation is the standard model of *loop unfolding*. The function f inside the loop is duplicated (or replicated some number of times) and the copies strung together in a series. The iteration then needs to cycle only some fraction of the number of times it otherwise would have. The intention is that, when the transformed program is loaded into the computer, the separate copies will execute simultaneously, except where data dependencies constrain them. For example, on a static data flow computer the operations of the different copies would be placed in different instruction cells that would fire independently. In an array processor the copies would be operated in lock-step by the different processors.

This loop unfolding is similar in concept to the "loop unrolling" somctimes used in conventional compilers [8, 14]. In conventional systems, however, the objectives are very different. There the aim is to reduce the number of times the exit test must be performed. There is rarely any assumption that the execution of the loop bodies will overlap at all.

The benefits of making separate copies of the transformation `f` can vary from virtually no overlap to complete overlap. On the surface, it might appear that, if we are computing `f(f(f(f(X))))`, the innermost function must complete before the next can begin, so one might as well leave the function in a loop. This is true for some functions, such as mathematical functions. If we need to compute `log(log(log(log(X))))`, we can't do the `log` operations in parallel. In a typical transformation, however, there is little or no *actual* data dependence among the evaluations of `f` because they refer to different parts of the set of loop variables. In particular, the different evaluations of `f` typically compute independent elements of a result array from different elements of an incoming array.

Any iteration made from a **forall** (and we expect that many iterations will be) necessarily has complete independence of the evaluations of `f`. Other iterations may not have complete independence, but will have near independence, with some sort of "critical path" encompassing only a small part of the computation.

3.1 Loop Unfolding

The standard unfolding is what was described previously, and will be called just *loop unfolding* or *loop expansion*. The *unfolding factor* is the number of copies of `f` that are expanded.

Here is unfolding factor 4:

```
%% compute f^N(INITVAL), N is multiple of 4
Z := for X, I := INITVAL, 0
do  if I=N then X
    else iter X, I := f(f(f(f(X)))), I+4 enditer
    endif
endfor
```

For simplicity, loop unfolding factors will always be powers of 2 in the examples. It is expected that this is the situation that will almost always arise in practice. If the unfolding factor (or the interlace factor, to be discussed later) is not a power of two, things become somewhat more difficult. This will be discussed in Section 4.6.

Since many iteration loops will come from **forall's**, it is useful to examine the loop unfolding of a **forall**.

```
%% compute Z[I] = g(I), 0 ≤ I < N, N is multiple of 4
Z := forall I in [0, N-1]
construct g(I)
endall
```

becomes

```
Z := for X, I := Λ, 0
do  if I=N then X
    else iter X, I := X[I: g(I)], I+1 enditer
    endif
endfor
```

which becomes, with unfolding of 4

```
Z := for X, I := Λ, 0
do  if I=N then X
    else iter X, I := X[I: g(I), g(I+1), g(I+2), g(I+3)], I+4 enditer
    endif
endfor
```

where X[I: α, β, γ, δ] is the VAL notation for an **append** that stores several values at consecutive indices. To get the full benefit of the unfolding, we assume that the 4 storage operations can be performed in parallel. Array interlace, described later, will accomplish this.

3.2 Initial Unfolding

Another transformation is called *initial unfolding*, and consists of taking some number of copies of f out of the loop altogether and doing them first. For example, we might do the first 3 cycles initially.

```
%% compute f^N(INITVAL), N ≥ 3
Z := for X, I := f(f(f(INITVAL))), 3
do  if I=N then X
    else iter X, I := f(X), I+1 enditer
    endif
endfor
```

3.3 Final Unfolding

The third transformation is called *final unfolding*: we execute the loop until some fixed number of cycles remain, and then do those cycles separately.

```
%% compute f^N(INITVAL), N ≥ 3
Z := for X, I := INITVAL, 0
do  if I=N-3 then f(f(f(X)))
    else iter X, I := f(X), I+1 enditer
    endif
endfor
```

We can combine all three types of unfolding in a single loop. The following has initial unfolding=3, loop unfolding=4, and final unfolding=2.

```
%% compute f^N(INITVAL), N = 5+some multiple of 4
Z := for X, I := f(f(f(INITVAL))), 3
do  if I=N-2 then f(f(X))
    else iter X, I := f(f(f(f(X)))), I+4 enditer
    endif
endfor
```

If the number of cycles that a loop will undergo is known, and we perform an initial or final unfolding by that number, the loop will disappear completely. This is because the loop that remains after the unfolding will have its end test satisfied immediately. A popular example of this is a **forall** with fixed limits.

```
Z := forall I in [0, 3]
construct f(I)
endall
```

This loop will have 4 cycles, which is a small enough number that we can perform a complete initial unfolding, obtaining

```
Z:= [0: f(0), f(1), f(2), f(3)]
```

3.4 Unfolding When the Number of Cycles is Not Well-behaved

Each of the transformations, as shown in the comments of the programs above, requires that certain properties of **N** be satisfied. It is expected that, when an optimizing compiler makes any of these transformations, it will know whether the requirements are satisfied. This is because **N** has been specified as a "parameter", in the manner described in Section 2.2. It is still possible, with extra code, to handle the general case. Consider initial unfolding. If we did not know that $N \geq 3$, we could write

```
Z := if N=0 then INITVAL
elseif N=1 then f(INITVAL)
elseif N=2 then f(f(INITVAL))
else
    for X, I := f(f(f(INITVAL))), 3
    do  if I=N then X
        else iter X, I := f(X), I+1 enditer
        endif
    endfor
endif
```

A similar sort of thing could be done for final unfolding. There is considerable waste of space, but the waste of time becomes relatively insignificant for large **N**.

For loop unfolding, if **N** is not the required multiple of the unfolding factor, we do this

```
%% unfolding = 4, but can handle any value of N
Z := for X, I := INITVAL, 0
do  if I ≤ N-4 then iter X, I := f(f(f(f(X)))), I+4 enditer
    else
        if I=N then X
        elseif I=N-1 then f(X)
        elseif I=N-2 then f(f(X))
        else f(f(f(X)))           % I = N-3 here
        endif
    endif
endfor
```

The waste of time in this should be comparatively small for large **N**.

If **N** is not a multiple of the unfolding but its remainder modulo the unfolding is known, we can do better than the above example. If `N mod 4 = 3`, we can use an initial (or final) unfolding of 3. What is left over will be a multiple of 4.

To summarize, it is only when the number of cycles is not known (as opposed to known but not a multiple of the unfolding) that we need to put in extra conditionals. Wherever possible, the compiler should know the number of cycles of the important loops in a program.

3.5 Loops That Have No Obvious Index Variable

The foregoing has been defined in terms of an index variable that counts up to a fixed limit. This was done because most of the loops that are important in numerical programs (including all loops arising from forall's) are of this type. However, loop unfolding, initial unfolding, and final unfolding can be performed on any loop at all.

A typical loop is

```
for X := INITVAL
do  if P(X) then R(X)
    else iter X := S(X) enditer
    endif
endfor
```

where X denotes all of the loop variables, P denotes the exit predicate, R denotes the value(s) to return, and S denotes the next-state function.

If we know that this will cycle at least 3 times, we can perform an initial unfolding of 3:

```
for X := S(S(S(INITVAL)))
do  if P(X) then R(X)
    else iter X := S(X) enditer
    endif
endfor
```

If we know that this will cycle a multiple of 4 times, we can perform a loop unfolding of 4:

```
for X := INITVAL
do  if P(X) then R(X)
    else iter X := S(S(S(S(X)))) enditer
    endif
endfor
```

We can similarly perform a final unfolding. These transformations depended on some knowledge of how many cycles would take place, in that they neglected to evaluate `P(X)` in certain places and just assumed `P(X)` was false. This is equivalent to the assumptions made previously about `N` when the loop was controlled by a counter. If we can't make assumptions about when `P(X)` will be true, then we have to put in extra code, just as in the loops controlled by a counter. For example, a completely safe initial unfolding would look like

```
if P(INITVAL) then R(INITVAL)
elseif P(S(INITVAL)) then R(S(INITVAL))
elseif P(S(S(INITVAL))) then R(S(S(INITVAL)))
else
    for X := S(S(S(INITVAL)))
    do  if P(X) then R(X)
        else iter X := S(X) enditer
        endif
    endfor
endif
```

3.6 Rescaling the Index Variable

When loop unfolding is performed on a loop controlled by an index variable, it is often useful (i.e. it saves excess arithmetic operations) to *rescale* the index variable, usually by dividing it by the amount of the unfolding.

If we have:

```
Z := for X, I := INITVAL, 0
do  if I=N then X
    else iter X, I := f(X), I+1 enditer
    endif
endfor
```

and we perform loop unfolding of 4:

```
Z := for X, I := INITVAL, 0
do  if I=N then X
    else iter X, I := f(f(f(f(X)))), I+4 enditer
    endif
endfor
```

we can introduce a new variable J = I/4 and rewrite this as

```
Z := for X, J := INITVAL, 0
do  if J=N/4 then X
    else iter X, J := f(f(f(f(X)))), J+1 enditer
    endif
endfor
```

This is particularly useful when interlaced arrays are involved.

Rescaling can also involve adding or subtracting a fixed offset. For example

```
Z := for X, I := INITVAL, 1
do  if I=97 then X
    else iter X, I := f(X), I+1 enditer
    endif
endfor
```

could expand to

```
Z := for X, I := INITVAL, 1
do  if I=97 then X
    else iter X, I := f(f(f(f(X)))), I+4 enditer
    endif
endfor
```

which could be rescaled by I = 4J+1, obtaining

```
Z := for X, J := INITVAL, 0
do  if J=24 then X
    else iter X, J := f(f(f(f(X)))), J+1 enditer
    endif
endfor
```

3.7 Interlace

Perhaps the most important transformation one can perform to improve the performance of numerical computations involving arrays is to spatially separate array operations that need to take place at nearly the same time. By having the array operations take place in different hardware units, the bottleneck is removed, and those hardware units can do their tasks in parallel. This is the basis for the memory interlace used in vector and pipeline computers, and the physically distinct processing units of array computers.

Arrays can be interlaced directly in the data flow graph. The amount of interlace for each array can be chosen independently for optimum performance. By interlacing arrays directly in the graph, the issues relating to interlace can be separated from issues of the computer hardware, though the optimum amount of interlace for a given program requires knowing how the machine behaves.

Interlace consists of dividing the array into α *slices*, where the slice to which an element belongs is determined by its index modulo α. α is the *interlace factor* (or the *interlace*). For simplicity it is always a power of 2.

If an array A is given an interlace factor of 4 (or a "4 way interlace") there are 4 slices, called A_0, A_1, A_2, and A_3. Those elements that were originally A[-8], A[-4], A[0], A[4] etc. are stored in A_0. Those elements that were A[-3], A[1], A[5] etc. are stored in A_1, and so on.

The mapping function for interlace factor α is

$$\underset{\text{(in the original array)}}{A[\alpha I+J]} = \underset{\text{(in the interlaced array)}}{A_J[I]}$$

$$\text{where } 0 \leq J < \alpha$$

or equivalently,

$$A[I] \text{ is stored in } A_{I \bmod \alpha}[\lfloor I/\alpha \rfloor]$$

where mod is defined to yield a nonnegative result and $\lfloor X \rfloor$ truncates toward $-\infty$.

3.8 Graphical Representation of Unfolding

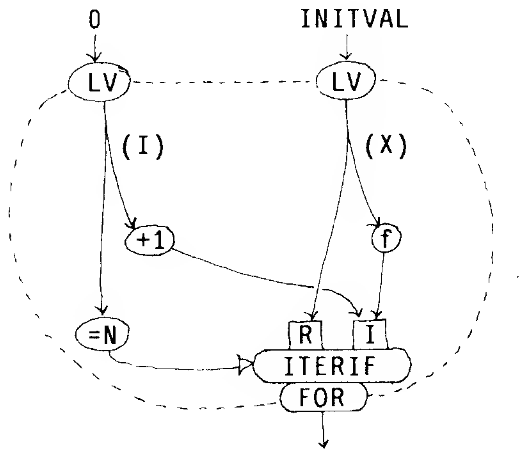
When iterations are drawn in graphical notation, loop unfolding is performed essentially by splicing together multiple copies of the part of the loop body that lies between the "LV" nodes and the values going into the "iter".

Referring back to the original example:

```
Z := for X, I := INITVAL, 0
do   if I=N then X
     else iter X, I := f(X), I+1 enditer
     endif
endfor
```

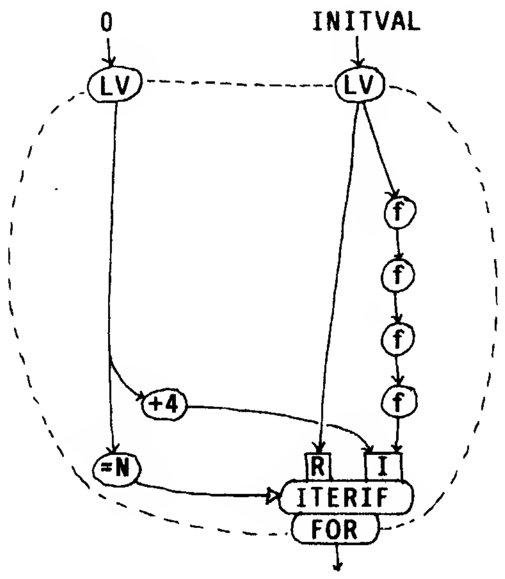
the graph is

Fig. 3.1



with 4-way unfolding, assuming we know that **N** is a multiple of 4:

Fig. 3.2



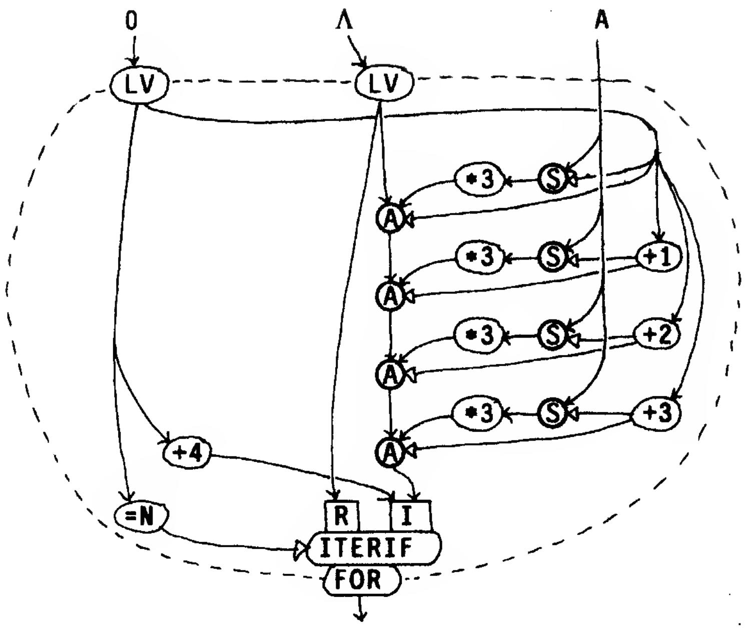
Of course, in a typical loop that arises in practice, the part that gets replicated is more than just the single node "f". It is a complex subgraph which, when replicated, overwhelms the rest of the loop. Furthermore, the replicated copies are often intertwined with each other quite intricately, as subsequent chapters will show.

When unfolding is performed on a loop making references to arrays, and the arrays are interlaced, the compiler can use its information about the array index in each unfolded instance to optimize the array references. Certain references in certain loop instances will, by virtue of the known array index modulo the interlace factor, refer only to certain array slices. The resulting graph typically looks something like this:

```
B := forall I in [0, N-1] construct A[I]*3 endall
```

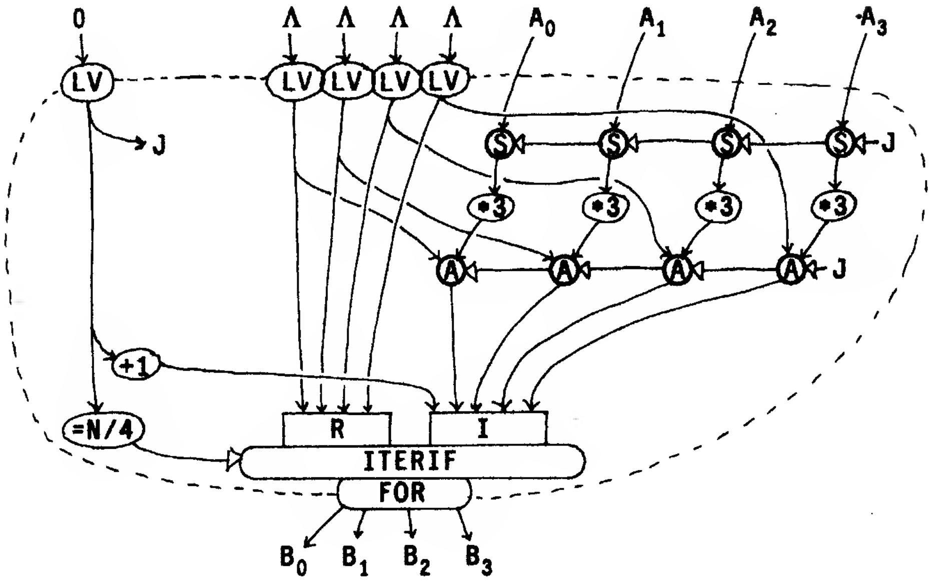
with unfolding 4 only:

Fig. 3.3



with interlace of 4 on both incoming and result arrays, and rescaling:

Fig. 3.4



The potential for parallelism when unfolding and interlace are used can be readily seen.

3.9 Nested Loops

When loops are nested one can choose the unfolding independently at each level.

Consider the following nested loop:

```
%% assume M and N are multiples of 4
for I, X := 0, INITVAL
do  if I = N then X
    else
        let NEWX := for J, Y := 0, X
            do  if J = M then g(Y, I)
                else iter J, Y := J+1, f(X, Y, I, J) enditer
                endif
            endfor ;
        in iter I, X := I+1, NEWX enditer
        endlet
    endif
endfor
```

The functions f and g symbolize the inner workings of the loops. Their dependence on the values of the counter variables is shown explicitly, which will demonstrate that loop unfolding requires a lot of careful bookkeeping.

If we perform a loop unfolding of 4 on the inner loop, we get the following, where some extra assignments have been made to keep the complexity manageable.

```
// this is the inner loop only
for J, Y := 0, X
do  if J = M then g(Y, I)
    else
        let NEWY1 := f(X, Y, I, J) ;
            NEWY2 := f(X, NEWY1, I, J+1) ;
            NEWY3 := f(X, NEWY2, I, J+2) ;
            NEWY4 := f(X, NEWY3, I, J+3) ;
        in iter J, Y := J+4, NEWY4 enditer
    endif
endfor
```

We can also perform a loop unfolding of 2 on the outer loop, obtaining

```
%% assume M and N are multiples of 4
for I, X := 0, INITVAL
do  if I = N then X
    else
        let NEWX1 := for J, Y := 0, X
            do  if J = M then g(Y, I)
                else
                    let NEWY1 := f(X, Y, I, J) ;
                        NEWY2 := f(X, NEWY1, I, J+1) ;
                        NEWY3 := f(X, NEWY2, I, J+2) ;
                        NEWY4 := f(X, NEWY3, I, J+3) ;
                    in iter J, Y := J+4, NEWY4 enditer
                endif
            endfor ;
            NEWX2 := for J, Y := 0, NEWX1
            do  if J = M then g(Y, I+1)
                else
                    let NEWY1 := f(NEWX1, Y, I+1, J) ;
                        NEWY2 := f(NEWX1, NEWY1, I+1, J+1) ;
                        NEWY3 := f(NEWX1, NEWY2, I+1, J+2) ;
                        NEWY4 := f(NEWX1, NEWY3, I+1, J+3) ;
                    in iter J, Y := J+4, NEWY4 enditer
                endif
            endfor ;
        in iter I, X := I+2, NEWX2 enditer
        endlet
    endif
endfor
```

This nested loop has "2*4" unfolding. (We could use different amounts of unfolding in the different instantiations of the inner loop arising from the outer unfolding, but there is rarely any reason to do so.) The total number of instantiations of the innermost loop body is the product of all of the unfoldings, 8 in this case. This product could be considered the "total" unfolding, since it is a measure of the total amount of space in the computer that the nested and unfolded loops will consume.

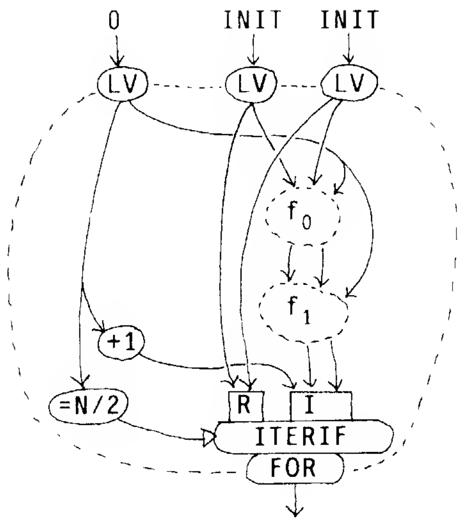
One might ask why we don't use all of the unfolding we have space for on the innermost loop, and none on the outer ones. On conventional computers, one typically optimizes the innermost loop most carefully. For example, variables manipulated by inner loops are given highest priority for using high-speed registers. We do not envision a data flow computer having high-speed registers that various parts of the computation compete for, so this consideration does not apply. When there are data dependencies from one iteration cycle to the next, preferentially unfolding the innermost loop provides a slight advantage in terms of overhead in loop control operators. However, when the loop cycles can be executed simultaneously, the loop cycles are correlated with elements of an array, and there are references to neighboring array elements in each unfolding, it is advantageous to unfold loops at all levels. The *shape* of the unfolding of nested loops refers to the relative amounts of unfolding at the various levels. It is the shape of the rectangular grid of unfolded loop cycles. This will be illustrated by various "2-dimensional" algorithms shown in Chapters 6 and 7.

3.10 Coalescing Control Structures

When there are nested loops, and unfolding is used, it very frequently occurs that the multiple instantiations of the inner loop can be executed simultaneously.

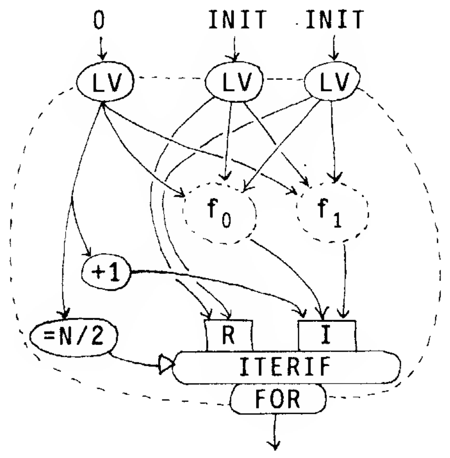
Suppose we use unfolding of two on some outer loop, obtaining something such as the following:

Fig. 3.5



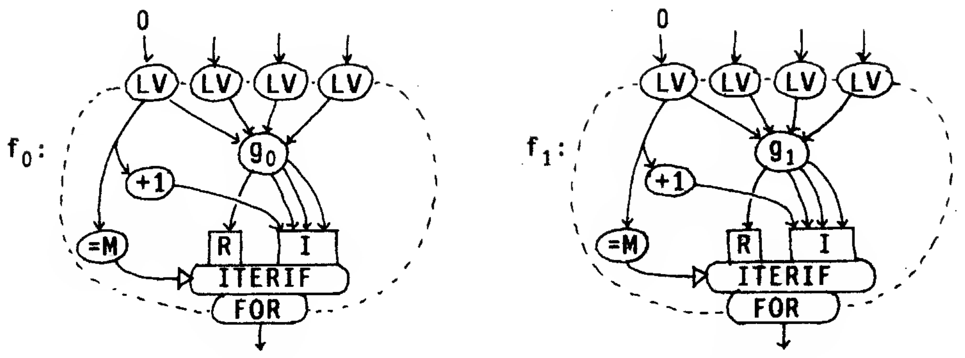
Now it may happen that, when the inner workings of the "f" subgraphs are worked out, there will in fact be no data dependency going from the first unfolding to the second. The graph might look like

Fig. 3.6



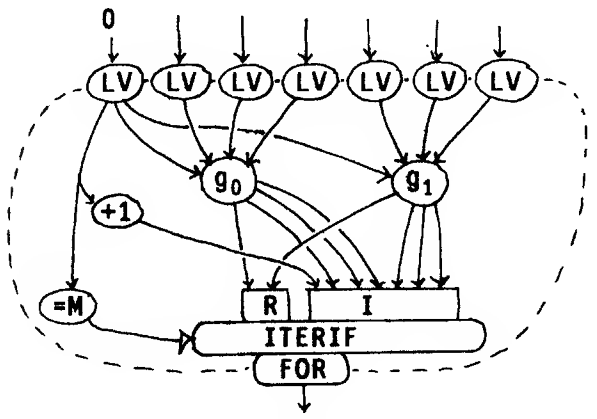
Now, if the occurrences of

Fig. 3.7



and the control structures are congruent, that is, the index variables and end tests are handled similarly, as is highly likely if the inner loops are just different instantiations of the same code arising from unfolding in the outer loop, they can be combined:

Fig. 3.8



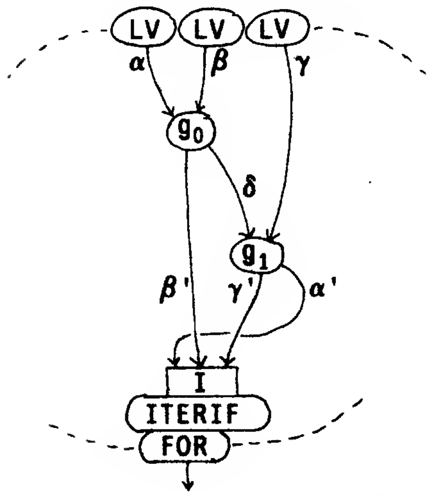
The benefit of this is not the fact that a few nodes have been saved, but that the subgraphs for g_0 and g_1, lying in the same loop, can share common subexpressions with each other. The significance of this will be discussed in Chapter 6.

One might think it unlikely that, when the outer loop is unfolded, the unfoldings would have no data dependencies among themselves, but in fact it occurs frequently. Almost all of the nested loops that will be examined in this thesis will have the control structures of the inner loops coalesced.

This coalescing of control structures is somewhat similar to the transformation known as "loop jamming" in conventional compilers [8, 14]. The transformation in conventional systems is performed in terms of "code motion" from one loop to another, principally for the purpose of economizing on loop control instructions. In the applicative case we have little concern for economy in the control computation, since it can be performed in parallel with the rest of the program. Instead, the objective is to prevent data values from having to be communicated among different loops, and thereby to allow common subexpressions to be combined.

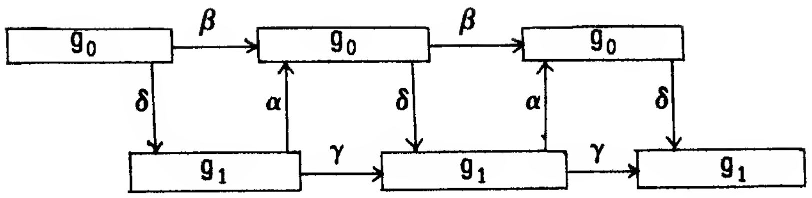
The fact that loop control structures are coalesced does not mean that they operate in lock-step. The recycling of loop variables from the "iterating arm" of an ITERIF back to the LV nodes should be able to occur at different times. The coalesced inner loop might look, in part, like

Fig. 3.9



Within each cycle, g_0 may have to be somewhat ahead of g_1. It should be possible for g_0's loop variable β to recycle and start the next cycle of g_0 while g_1 is still busy. The timing might look sort of like this, showing when each loop variable gets recycled (except δ, which is just passed from g_0 to g_1).

Fig. 3.10



This "decoupling" phenomenon will show up in the "wavefront transformation" in Section 10.1.

3.11 Interlace of Multidimensional Arrays

The model of multidimensional arrays we use is the "vector of vectors" model. Every genuine array in the language is actually a one-dimensional structure, whose element types can be anything. If we choose arrays of reals as the element type, we get an array of arrays of reals, which is equivalent to a two-dimensional array. If `X` is such an array, `X[I][J]` selects an element from it. `X[I]` selects the I^{th} one of the "second level" arrays, and `(X[I])[J]` selects the final element. This is the model used by some languages, such as VAL; CLU, and C, and one of the models available (indirectly) in most languages with a general type system, such as PASCAL.

The model used by some other languages, notably FORTRAN and PL/I, is the "flat" model. The lowest level rows are concatenated end-to-end to make one-dimensional arrays. If there are more levels, these results are then concatenated, and so on. The result is a one-dimensional array with a mapping function for references. For example, a [1-10]x[1-100]x[1-20] array (as might be declared in FORTRAN by "`DIMENSION A(10, 100, 20)`") is actually implemented as a single array running from 1 to 20000, and a reference to `A(I, J, K)` is treated as `A(2000*I + 20*J + K-2020)`.

From an implementation standpoint, flattening is a good idea. It makes all array selections possible with one memory reference (and array memory references may be expensive in data flow computers). It also means that only scalar data are actually stored in the array memory -- not array descriptors. This is extremely important in an applicative data flow system, because there is large overhead associated with reference accounting as array descriptors are manipulated [2, 3]. We want to cut down on manipulation of array descriptors, as will be made clear in Chapter 15, so we refrain from storing them in other arrays.

However, premature flattening of arrays will preclude the use of interlace to the same advantage in the multidimensional case as in the one-dimensional case. Therefore, we will use the "vector of vectors" model. When considering interlace and various program optimizations, arrays will be treated as true hierarchical objects. They will not be flattened. Interlacing will be performed on each dimension independently, yielding a multidimensional collection of slices. After loop unfolding and array interlace are performed, *each slice* will be flattened. The result will be far superior in general to what would be obtained by initially flattening the array and then interlacing the result.

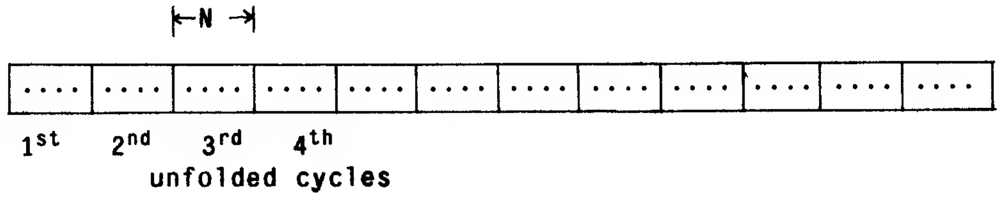
This multidimensional interleaving and flattening of individual slices, with each array considered separately, constitutes a radical restructuring of the arrays as described in the source program. When a complicated program is running on a data flow computer, the array elements will be scattered throughout the machine in a manner that is far from obvious. It is the applicative nature of the programming system that makes it feasible for a compiler to do this, because the compiler knows where every element of every array comes from.

Incidentally, the flattening of arrays in FORTRAN is mandated by the semantics. If `A` is a 10x10 array, it is legal to send its first row to a procedure `F` with the call `F(A(1))`, assuming `F` expects a one-dimensional, ten element argument. The second row may be sent with the call `F(A(11))`, and the K^{th} with `F(A(10*K+1))`. This works because the flattening mechanism is a fixed part of the language. It is therefore extremely difficult to write a FORTRAN compiler that uses some other array organization. This is a case of overspecification of the semantics of a programming language, an all too common occurrence. Note that, with the "vector of vectors" model, it is equally easy to send the K^{th} row of a two-dimensional array to some procedure. A data flow computer using the "vector of vectors" model has the freedom to interlace, flatten, or otherwise manipulate the array in many ways while remaining totally faithful to the language semantics.

3.12 Geometrical Representation of Unfolding

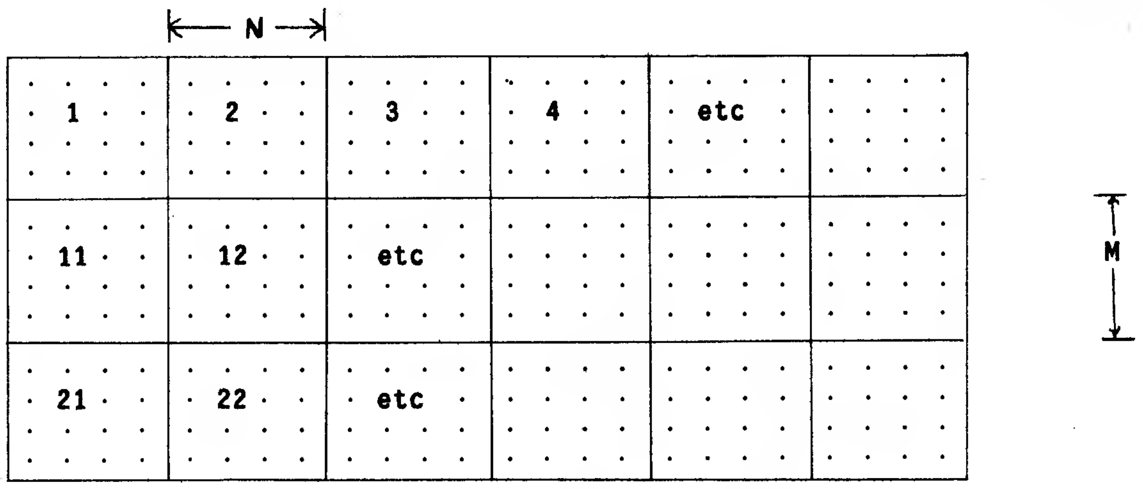
If we consider each cycle of a single loop to be a point, and perform **N**-way unfolding on that loop, then each unfolded cycle will process a group of **N** adjacent points. The groups will be processed in sequence.

Fig. 3.11



If we consider a double nested loop to be a plane array of points, the programmer's intention is that the points be processed sequentially in a "raster scan" order. Individual points in each row will be processed from left to right, and the rows will be processed from top to bottom. If we perform an M*N unfolding and coalesce the control structures of the inner loops, we will still have a double nested loop. Each cycle of that loop will process an M*N rectangle of points. The order of the cycles of the unfolded loop is such that the rectangles will be processed once again in a raster scan order, but on a coarser scale.

Fig. 3.12



For K-fold nested loops, each unfolded cycle will process a K-dimensional box in the K-dimensional space, and the boxes will be processed in the usual order.

This is the computational process we envision for a high-performance applicative computer. It will processes the data space in large localized chunks. It will work best on algorithms with locality of data dependence.

4. MATCHING INTERLACE AND UNFOLDING

This chapter will examine the relationship among the loop unfolding factor, the array interlace factor, and the parameters of the program for loops that deal with arrays. As will be seen, a careful match among these quantities is crucial to efficient translation. One of the major tasks of an optimizing compiler is to examine the parameters of the loops and choose the appropriate amounts of interlace for the arrays and unfolding for the loops. When this is done correctly, the resultant code structures will permit extremely rapid, bottleneck-free distributed computation.

The discussion will be primarily aimed at **select**'s. **Append**'s will also be considered, but in a simple-minded way, as though **append** were a slight variation of **select**. This assumption will suffice for the purposes of the present chapter, though there are other aspects of **append** that will be discussed in Chapter 8.

The situation that can be exploited well, in terms of parallel or overlapped accesses to interlaced arrays, is an iteration (or equivalent **forall**) whose array accesses are affine. The following example shows such an iteration. Any iteration in which all array accesses are affine is of this basic form.

```
for T, U := init_arrayT, init_arrayU ;
    <other initializations>
    J := init_index ;
do  if J=exit_index then T, U, <other values>
    else
        let V1 := ... expression ... A[J*factorA+offsetA] ... ;
            V2 := ... expression ... B[J*factorB+offsetB] ... ;
            ...
            new_Tvalue := ... expression ...
            new_Uvalue := ... expression ...
        in  iter
                T := T[J*factorT+offsetT: new_Tvalue] ;
                U := U[J*factorU+offsetU: new_Uvalue] ;
                <other reassignments>
                J := J+increment ;
            enditer
        endlet
    endif
endfor
```

Assume for now that all of the relevant parameters ("init_index", "factorA", "offsetA", "increment", etc.) are known to the compiler. If any are not known at compile time, it may still be possible to achieve extremely high performance, but it may require more complex code and more work on the part of the compiler. This will be discussed below.

The interesting activities in this loop are the array reads (of A and B) and writes (to T and U). These occur at regular intervals across the respective arrays. By examining the "factor" and "offset" of each array reference, and the initial index and increment, we can determine at just what index the array references occur. The *reference interval* is the distance along the array of the references in consecutive loop cycles. For each array it is just the "factor" for that array reference multiplied by the "increment" for the loop index variable. For example, the first reference to array A occurs at (factorA*init_index+offsetA), and the reference interval is (factorA*increment). It is possible for the same array to be referred to in more than one place in the same loop with different reference intervals.

Assume for simplicity that all reference intervals (as well as all unfolding factors and interlace factors) are powers of 2.

4.1 Random Array Access and the "PERMUTE" Operator

If a completely random access needs to be made to an interlaced array, that is, we have no information in advance about the index, the **select** or **append** node must be able to operate on any of the slices. If the array has an `N`-way interlace, the following is needed:

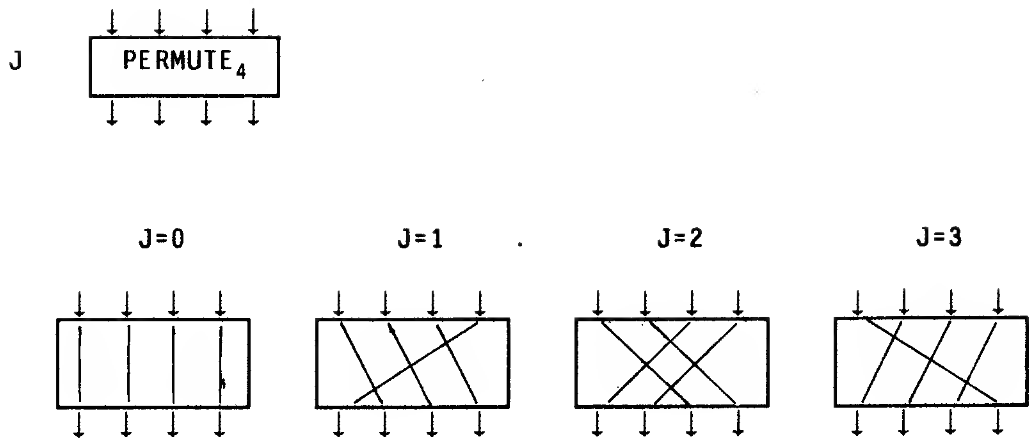
Fig. 4.1

to get	use
`A[J]`	$$\begin{tikzcd} & A_{J \bmod N} \arrow[d] \\ J/N \arrow[r] & \textcircled{S} \arrow[d] \\ & {} \end{tikzcd}$$

where the division operator is assumed to truncate toward minus infinity.

To get the slice "$A_{J \bmod N}$", a **permute** operator will be used. There is a **permute** operator for every value of `N` that is a power of 2. It permutes `N` incoming tokens by "rotating" them right the amount of its index value.

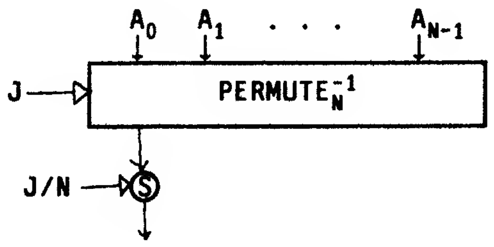
Fig. 4.2



The **permute**$^{-1}$ operator rotates left. We assume that the hardware has the necessary instruction types to do this efficiently, and that there will not be reference counting overhead if array tokens are permuted. (The reference counting problem for array tokens will be discussed in Chapter 15.) **Permute** operators are expensive, however. The large number of arguments indicates that, on any reasonable computer, a **permute** would have to be made out of a large number of actual instructions. One way to build it would be to use a network of 2*2 exchange instructions, built into a network such as an N-cube [43]. Each exchange instruction would be controlled by one bit of the binary representation of the index.

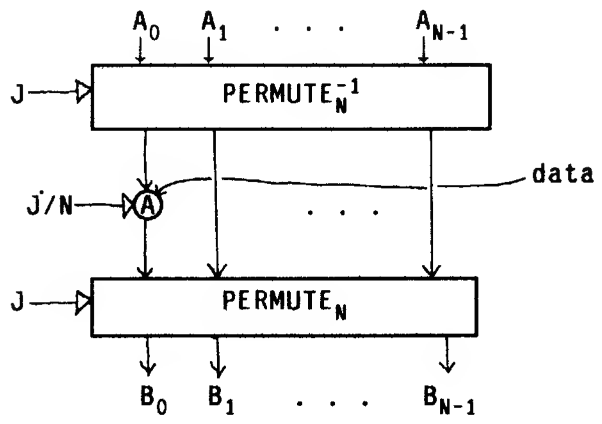
The **permute** operators will be assumed to use their index value modulo **N**. To get **A[J]**, use

Fig. 4.3



An **append** requires that the result be put back into the correct array slice, so it requires

Fig. 4.4



4.2 Affine Array Operations with Reference Interval Equal to One

It is clearly desirable to avoid the full "random" access, with its **permute** operator, presented in the previous section. In Chapter 3 it was shown that, if the desired index is the loop counter variable, the initial value the the counter variable is zero, and the interlace is equal to the unfolding, the situation is very nice. Each select or **append** operation in the expanded loop is known to refer to a specific slice, so no **permute** operations are ever needed. The index value going into the **select** or **append** is just the scaled counter variable.

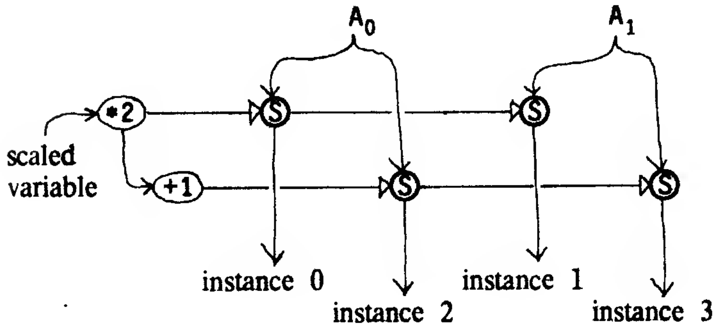
The first array reference does not necessarily take place at index zero, but at "`factor*init_index+offset`". The effect of this is to change the "phase" of the **N** array slices going into the **N** select or **append** operators. If this value is known at compilation time (or is known modulo **N**), a simple rearrangement of the incoming and outgoing array slices will take care of things. If the value is not known, some **permute** operators are needed to effect the rearrangement. These operators have to do their work only once, when the array slices enter the loop (and, in case of **append**, once again when they leave), so this is not too great a hardship. Handling the initial value of the loop counter variable and the array reference offset is therefore a relatively minor matter.

The important things to deal with are the reference interval, interlace, and unfolding. For now, let the reference interval be fixed at one.

If the unfolding is greater than the interlace, say twice as great, then two instantiations will have to share the same slice. One will use the slice at index equal to the scaled loop variable times two, and the other at that plus one.

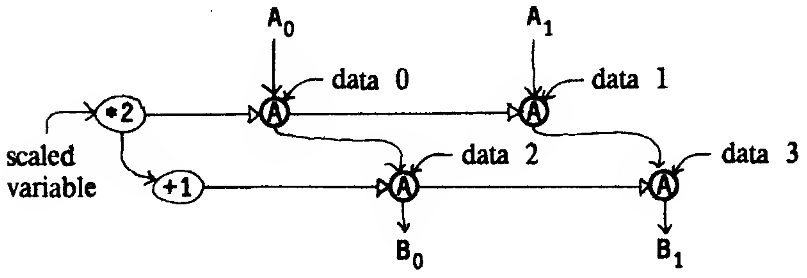
For unfolding = 4 and interlace = 2, we have

Fig. 4.5



Each slice goes into 2 **select** or **append** nodes (or more generally, into $\frac{\texttt{unfolding}}{\texttt{interlace}}$ of them). In the case of **append** nodes, the slice goes into them in series, as in

Fig. 4.6



The general rule for `unfolding` > `interlace` is that no **permute** operators are ever used, and that each array slice goes to $\frac{\texttt{unfolding}}{\texttt{interlace}}$ operators.[1] This situation isn't too bad, though the use of several **select**

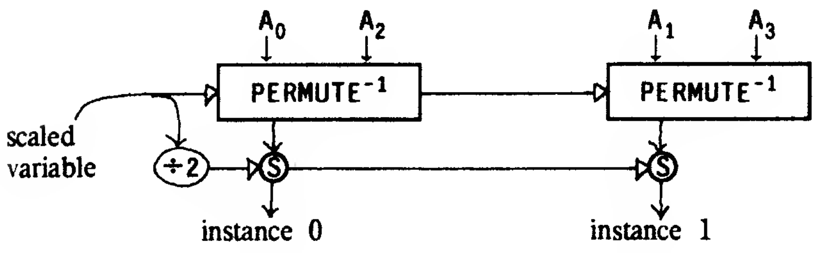
1. $\frac{\texttt{unfolding}}{\texttt{interlace}}$ is, of course, a power of 2.

or **append** operators on the same array slice may tend to create a bottleneck.

Now if the interlace is greater than the unfolding, say twice as great, then each instantiation will have to use two slices alternately, depending on whether the scaled variable is even or odd. The array index that it will present to the slice will be the scaled variable divided by two (perhaps with some offset added).

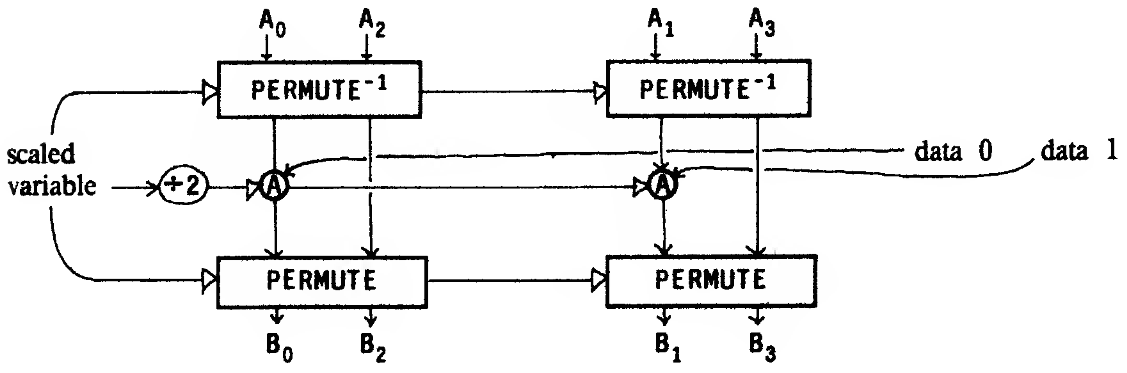
For unfolding = 2 and interlace = 4, we have

Fig. 4.7



Each pair of slices (or more generally, group of $\frac{\texttt{interlace}}{\texttt{unfolding}}$ of them) goes through an inverse permuter into one **select** or **append** node. In the case of **append** nodes, the result has to be permuted back. All the slices that do not go through the **append** node are passed directly from the **permute**$^{-1}$ to the **permute**.

Fig. 4.8



The general rule for `unfolding<interlace` is that each **select** or **append** operator in an instantiation needs a **permute**$^{-1}$ of size $\frac{\texttt{interlace}}{\texttt{unfolding}}$ to find the correct slice. The use of `PERMUTE` operators looks like something to be avoided, though it might be possible to build a computer in which they are not too expensive.

4.3 Reference Interval Not Equal to One

Suppose that the reference interval is not equal to one, but is some other number (a power of 2, of course) known at compilation time. The important criterion determining what will happen is the comparison of the following two values:

```
reference interval * unfolding  :  interlace
```

If the reference interval times the unfolding is smaller than the interlace, **permute** operators will be needed, with $\frac{\texttt{interlace}}{\texttt{reference interval*unfolding}}$ inputs. If larger, each array slice will go to, approximately, $\frac{\texttt{reference interval*unfolding}}{\texttt{interlace}}$ **select** or **append** nodes. (A correct version of this number will be given later.) Note that the previous results for reference interval equal to one agrees with this.

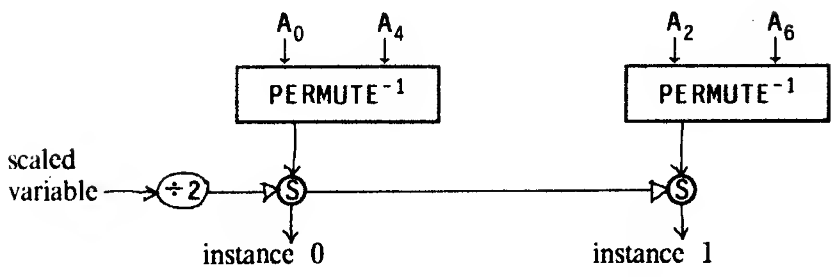
To analyze these cases, consider first the effect of a reference interval less than the interlace. Suppose the array A uses an interlace of 8, so that it is divided into slices A_0, A_1 ... A_7. If an iteration refers to it with a reference interval of two, say only even indices, then only A_0, A_2, A_4, and A_6 are actually used. Inside the loop, the array appears to have only four slices, that is, only a 4-way interlace. Note further that *every* element of each slice is referred to, instead of every other element. The array appears inside the loop to have a reference interval of one, not two. In general, the array appears to have an "effective interlace" equal to $\frac{\texttt{true interlace}}{\texttt{reference interval}}$. Since the loop appears to have a reference interval of one, the preceding section applies.

4.4 Reference Interval * Unfolding < Interlace --- Permute

In this case, the reference interval is clearly less than the interlace. The loop behaves as though the "effective interlace" were $\frac{\texttt{interlace}}{\texttt{reference interval}}$. This is greater than the unfolding, so there are **permute** operators with $\frac{\texttt{interlace}}{\texttt{reference interval*unfolding}}$ inputs.

Example: reference interval = 2, unfolding = 2, interlace = 8

Fig. 4.9

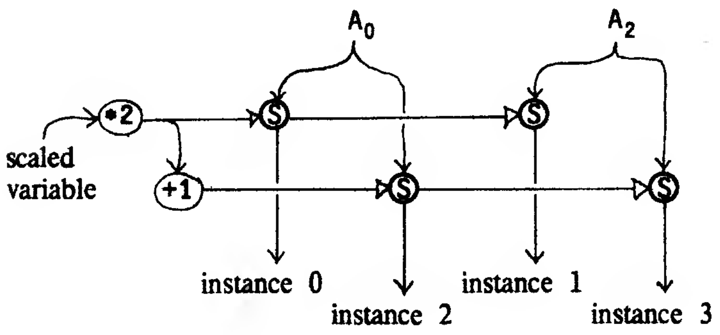


4.5 Reference Interval * Unfolding > Interlace --- Multiple Operators

In this case, we must compare the reference interval and the interlace. Suppose the reference interval is smaller. Then the loop behaves as though the reference interval were one and the "effective interlace" were $\frac{\texttt{interlace}}{\texttt{reference interval}}$. Since the latter quantity is less than the unfolding, each array slice that is used goes to $\frac{\texttt{reference interval*unfolding}}{\texttt{interlace}}$ **select** or **append** operators.

Example: reference interval = 2, unfolding = 4, interlace = 4

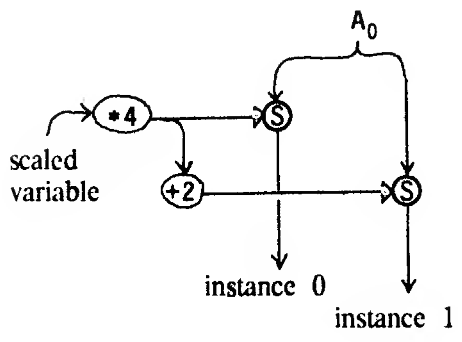
Fig. 4.10



Now suppose the reference interval is greater than the interlace. Then only one slice is actually used, and the "effective reference interval" for the index values going into the array operators is $\frac{\texttt{reference interval}}{\texttt{interlace}}$. Since there is only one slice, there are no **permute** operations. That slice goes to every **select** or **append** operator -- one per instantiation, so each slice that is used is used in a number of operators equal to the unfolding.

Example: reference interval = 4, unfolding = 2, interlace = 2

Fig. 4.11



To summarize:

$\boxed{\text{reference interval} * \text{unfolding} < \text{interlace} :}$

$\frac{\text{interlace}}{\text{reference interval}}$ slices are used; there is a permuter of $\frac{\text{interlace}}{\text{reference interval}*\text{unfolding}}$ inputs.

$\boxed{\text{reference interval} * \text{unfolding} > \text{interlace} :}$

$\left\lceil \frac{\text{interlace}}{\text{reference interval}} \right\rceil$ slices are used;

each one that is used goes to $\frac{\text{unfolding}}{\left\lceil \frac{\text{interlace}}{\text{reference interval}} \right\rceil}$ places.

The general rule of thumb is that the reference interval times the unfolding should be equal to the interlace, so that there are neither permutes nor array slices passing through many array operators. This rule may not always be exactly right in practice. Various factors involving the machine speed and the nature of the computation may dictate a ratio other than unity. However, the best ratio will not be grossly different from unity, and will stay reasonably constant as the machine size scales upward (that is, it will not go asymptotically to zero or infinity). If a unit ratio is not obtainable, it is better to have the reference interval times the unfolding be greater than the interlace, to avoid the necessity for permuters.

What happens if the reference interval is not known to the compiler? All is not lost. It simply becomes more difficult to take advantage of the interlace. At worst, all references become "random accesses". Of course, we want to avoid brute-force random accesses if at all possible. There are several useful cases in which they can be avoided. If the reference interval is known to lie within a small bound (typically it is either zero or one, depending on run-time circumstances), a small amount of messy object code can run quite efficiently. This is discussed in Chapter 9. Another case of interest is the one in which the reference interval, while unknown, is known to be a multiple of the interlace factor. In this case the interlace does not cause any inconvenience at all. An example of this is the inner loop of the "cyclic reduction algorithm", discussed at length later in this chapter. More generally, whenever the reference interval and the interlace factor have a known common divisor greater than one, we can do better than brute-force access.

4.6 When Things Are Not Powers of Two

The preceding treatment depended on the fact that if X and Y are both powers of the same prime, either X divides Y or Y divides X, depending on which number is smaller. This made it possible to say that perhaps permuters would be needed, or perhaps array slices would be used in multiple loop instances, but not both. If the interlace, unfolding, or reference interval are not powers of two (or some other prime) we might have the worst of both worlds -- slices used in multiple loop instances and permuters required in each instance. In particular, it is not possible to remove the need for permuters simply by increasing the unfolding, unless the factorization of the unfolding number is taken into consideration.

The "effective interlace", that is, the number of array slices that are actually used, is given by

$$\mathrm{USE} = \frac{\mathrm{interlace}}{\mathrm{GCD}(\mathrm{interlace},\ \mathrm{reference\ interval})}$$

This is in all cases consistent with the formulas given previously, because of the equation

$$\frac{X}{\mathrm{GCD}(X,\ Y)} = \left\lceil \frac{X}{Y} \right\rceil \quad \text{if X and Y are both powers of the same prime}$$

Each slice goes to $\frac{\texttt{unfolding}}{\texttt{GCD(unfolding, USE)}}$ loop instances

Each loop instance requires a permuter of $\frac{\texttt{USE}}{\texttt{GCD(USE, unfolding)}}$ inputs

It was shown previously that, given the interlace and the reference interval, one could remove the need for permuters simply by making the unfolding large enough -- it had to be at least $\frac{\texttt{interlace}}{\texttt{reference interval}}$. If the numbers are not all powers of two, things are slightly more difficult. We need to make `USE = GCD(USE, unfolding)`, which requires that the unfolding be an integral multiple of `USE`, that is,

$$\texttt{unfolding} = \text{a multiple of } \frac{\texttt{interlace}}{\texttt{GCD(interlace, reference interval)}}$$

4.7 Loops with Many Arrays

Loops may well have several arrays coming in or going out, and many aspects of the program interact when we choose the interlace for each array and the unfolding for each loop. In making these choices, a few obvious truths should be noted.

1. The unfolding in any loop must be one value, not different values in different parts of the loop. If a loop's use of one array suggests that an unfolding of four is optimal and its use of another array suggests eight, a consistent decision must be made. Typically the larger value will be used, since that will avoid the use of **permute** operators.

2. The interlace of any one array must be one value that is consistent for the array's producer and all of its consumers. When matching the interlace of an array to the various loops in which it is used, it is generally best to choose the smallest of the "optimal" values that the loops suggest, once again to avoid **permute** operators. (Actually, one could run an array through an "interlace converter" if a mismatch threatened to be serious.)

3. The reference interval for any array reference in a loop is a property of the program and not under the optimizer's control. The same array could be used in multiple places in the same loop with different reference intervals, that is, the reference interval is a property of each *reference*, not each array.

4.8 Choosing the Interlace and Unfolding

The method for making the various choices of array interlace and loop unfolding might be roughly as follows. It is proposed that, until really sophisticated techniques become available, a computer program to perform these transformations use human-generated "advice", either by querying the user when running or by reading a file.

We first determine, for each loop, what amount of loop unfolding to use. This depends on a compromise between the amount of space available in the machine for instruction cells (large unfolding in large loops consumes an enormous amount of space) and the benefit to be derived from the unfolding. Many factors interact in the unfolding decisions for the various loops in the program. We will discuss some of the issues later, but a truly "scientific" analysis of the problem has not been developed.

Having determined unfolding, we determine the interlace for each array roughly as follows. For each loop that creates or consumes an array, we know the reference interval, and the "ideal" interlace is the product of the reference interval and the unfolding. In many cases, these "ideal" interlaces won't all be the same. If this happens, we generally use the smallest figure. This avoids **permute** nodes but means that we may have array slices going to more than one **select** or **append** node.

The interaction between the loop unfolding and the array interlace does not go in one direction only: We may choose the unfolding of a loop to match the interlace of an array, instead of the other way around. This will happen when two or more loops are linked to each other through an array, and the speed of one of the loops makes it pointless to use as much unfolding in the other as we would like.

An optimizer that automatically chooses interlace and unfolding without human advice will have to be able to handle all of the "local" factors considered here, and "global" factors as well. When an array is created in one loop and used in one or more other loops, the ideal amounts of unfolding in the various loops interact through the array interlace. The individual choices must therefore be made in the presence of many constraints, some of which are likely to be inconsistent. The job of the optimizer is to make these choices, making compromises where necessary, in a way that is compatible with the overall size of the computer and will yield the best performance.

4.9 The Periodic Cyclic Reduction Algorithm -- Reduction Part, Inner Loop

The following program illustrates the considerations that go into choosing unfolding and interlace. It is the inner loop of the "reduction" part of the "periodic cyclic reduction algorithm" (also known as "even-odd reduction") for solving certain second order differential equations in one dimension with periodic boundary conditions. The equation is

$$X'' + AX = Q$$

That is, given the array Q and the scalar A, it finds the array X that solves the difference equation

$$X[J+1] + X[J-1] + (A-2)*X[J] = Q[J] \qquad 2 \leq J \leq SIZE+1$$

subject to the boundary conditions `X[1] = X[SIZE+1]` and `X[2] = X[SIZE+2]`.

The complete algorithm is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% Periodic Cyclic Reduction
%%
%% Produces array X[2..SIZE+1] such that, for i=3..SIZE,
%%
%%   X[i+1] + X[i-1] - 2*X[i]
%%   ------------------------ + A * X[i] = - Q[i]
%%           hx**2
%%
%% That is, del-squared(X) + A*X = -Q in one dimension.
%%
%% Boundary conditions "wrap around": The above equation is true
%%   for i=2, but with X[SIZE+1] in place of X[1], and is true for
%%   i=SIZE+1, but with X[2] in place of X[SIZE+2].
%%
%% A must not be zero.
%%
%% SIZE must be a power of 2, and at least 4.
```

```
function PERI1(Q: areal; HX, C: real; SIZE: integer returns areal)
type areal=array[real] ;

let
    %% Scale incoming array for mesh size.
    FAC := HX*HX ;
    QA: areal := forall i in [2, SIZE+1] construct Q[i]*FAC endall ;

    %%%%%%%%%%%%%%%
    %% REDUCTION %%
    %%%%%%%%%%%%%%%

    LOGSIZE, FINA, FINQ: areal, FINB: areal :=
    for IH := 1 ;                          % IH runs from 1 through SIZE/2
        COUNT := 1 ;
        B: areal := array_empty[real] ;
        A := 2.0-C*FAC ;
        Q: areal := QA ;                   % QA is the scaled incoming array
    do
        let
            %% Produce an array T with
            %% T[K] = Q[K-IH] + A*Q[K] + Q[mod(K, SIZE)+IH]
            %% for all K = 1 + a multiple of 2*IH

            NEWQ :=
            for K := 2*IH+1 ;   % IH is a power of 2
                T := Q ;        % Q, T bounds are 2 to SIZE+1 inclusive
                QK := Q[IH+1] ;
            do  if K > SIZE+1 then T
                else
                    let NQK := Q[mod(K, SIZE)+IH]
                    in  iter K, T, QK :=    % happens SIZE/(2*IH) times
                            K+2*IH,
                            T[K: QK + A*Q[K] + NQK],
                            NQK
                        enditer
                    endlet
                endif
            endfor ;

            NEWB: areal := array_addh(B, A) ;
        in
            if IH = SIZE/2 then
                COUNT, A*A-4.0, NEWQ, NEWB
                %% the A*A-4.0 makes the wrap-around work,
                %%    it will lose if C=0
            else
                iter IH, COUNT, B, A, Q :=
                    IH*2, COUNT+1, NEWB, A*A-2.0, NEWQ enditer
            endif
        endlet
    endfor ;

    %% Repair last element with scale factor magically computed above.
    QB: areal := FINQ[SIZE+1: FINQ[SIZE+1]/FINA] ;
in
    %%%%%%%%%%%%%%%%%%
    %% SUBSTITUTION %%
    %%%%%%%%%%%%%%%%%%

    for IH := SIZE/2 ;                  % IH runs from SIZE/2 to 1
        LOGI := LOGSIZE ;
        Q: areal := QB ;
    do  let A := FINB[LOGI] ;
            NEWQ :=
            for K := IH+1 ;
                QK := Q[SIZE+1] ;
                T := Q ;
            do  if K > SIZE+1 then T
                else
                    let NQK := Q[K+IH]
                    in
                        iter K, T, QK :=
                            K+2*IH,
                            T[K: (QK + Q[K] + NQK)/A],
                            NQK
                        enditer
                    endlet
                endif
            endfor

        in  if IH = 1 then NEWQ
            else iter IH, LOGI, Q := IH/2, LOGI-1, NEWQ enditer
            endif
        endlet
    endfor
endlet
endfun
```

For now, we will just look at the inner loop of the reduction part (the first half of the algorithm). Because there are two nested loops, there are, in effect, several nearly identical inner loops, that differ in the value of IH, which is a loop variable of the outer loop. IH is always a power of two, varying from 1 up to SIZE/2. SIZE (also a power of 2) is the size of the matrix being solved, and is very large (perhaps thousands of elements). Since the reference interval of the inner loop is IH*2, the analysis varies from one cycle of the outer loop to the next. Hence we consider each value of IH as defining a different inner loop. The array comes in as Q. The result is built as the loop variable T, and is yielded as the final result of the inner loop. The result of one inner loop goes into the next one as Q, so the interlace of Q and the result should be compatible from one inner loop to the next. Since the reference interval for the select references to Q and for the **append** references to the result are both the same, the "ideal" interlaces are the same, so we might as well use the same interlace for everything. The bounds of Q and T are [2, SIZE+1] inclusive.

The inner loops are as follows:

```
for K := 2*IH+1 ;              % IH is a power of 2
    T := Q ;                   % Q, T bounds are 2 to SIZE+1 inclusive
    QK := Q[IH+1] ;
do  if K > SIZE+1 then T
    else
        let NQK := Q[mod(K, SIZE)+IH]
        in  iter K, T, QK :=       % happens SIZE/(2*IH) times
                K+2*IH,
                T[K: QK + A*Q[K] + NQK],
                NQK
            enditer
        endlet
    endif
endfor
```

The above code is not the way it really ought to be written. The loop variable QK was manually introduced to reduce the number of array references. This should be done by an optimizer, which will be discussed in Section 7.4.

There are two **select** references to Q within the loop, and one **append** reference to T. They all have reference interval 2*IH,[1] which is a variable with respect to the outer loop, but is, fortunately, a constant within any one of the inner loops. The number of inner loops is $\log_2$(SIZE).

1. One of the references has a mod operator in its index. The reference interval is still 2*IH, but an optimizer will have to work hard to deduce that. We propose that optimizers in fact be able to do so. The computation could be expressed with an if/then/else instead of **mod**, which would also require that an optimizer work hard.

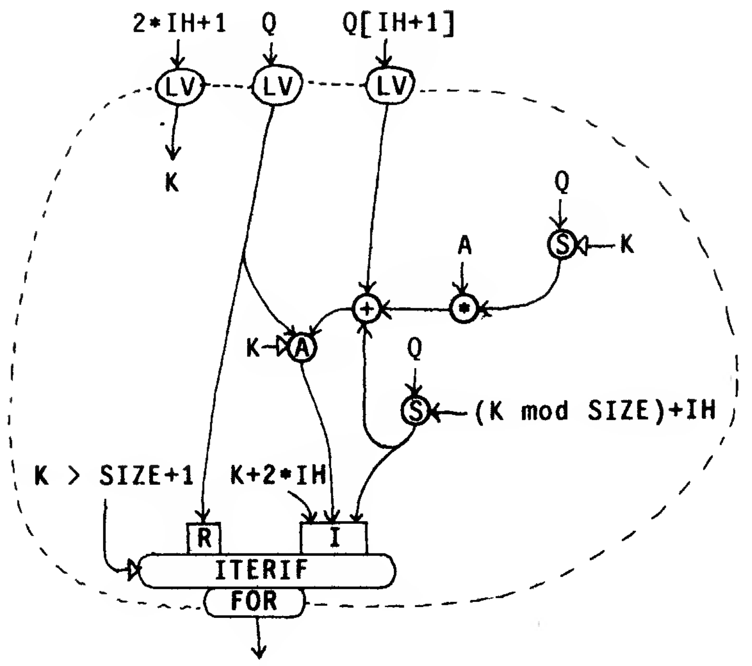
Suppose that, on the basis of some obscure analysis, we choose an interlace of 16. The "ideal" unfolding will vary from one loop to another because of the changing reference interval, as follows:

IH	reference interval	"ideal" unfolding	number of virtual cycles	number of expanded cycles
1	2	8	SIZE/2	SIZE/16
2	4	4	SIZE/4	SIZE/16
4	8	2	SIZE/8	SIZE/16
8	16	1	SIZE/16	SIZE/16
16	32	1	SIZE/32	SIZE/32
32	64	1	SIZE/64	SIZE/64
.	.	.	.	.
.	.	.	.	.
IH	2*IH	1	SIZE/(2*IH)	SIZE/(2*IH)
.	.	.	.	.
.	.	.	.	.
SIZE/2	SIZE	1	1	1

For the first 4 loops, the criterion `reference interval*unfolding = interlace` is met exactly. No **permute** operators are needed in any of the loops.

This is the basic graph:

Fig. 4.12



Those loops for which IH is 8 or more have the reference interval a multiple of the interlace factor. As noted previously, this means that the fact that we don't know the reference interval is no problem. Because of this, along with the fact that these loops have the same unfolding, they can all be treated as one loop with different parameters. The cases IH=1, 2, and 4 must be removed and treated as special cases. This is not surprising -- since they have different unfoldings they will have very different structures and will have to be compiled separately. The case IH=8 should also be treated differently if **permute**'s are to be avoided. The select required in "Q[mod(K, SIZE)+IH]" has reference interval a multiple of 16 whenever IH $\geq$ 8, but the offset (the "IH") is only 8 when IH is 8, which would require a **permute**. The initial value for QK, which is Q[IH+1], has a similar problem.

So, to avoid any permutes, only the cases with IH $\geq$ 16 will be made into a common loop. The arrays Q and T are interlaced 16 ways.

The counter variable K is rescaled by

$$K_{old} = 16 * K_{new} + 1$$

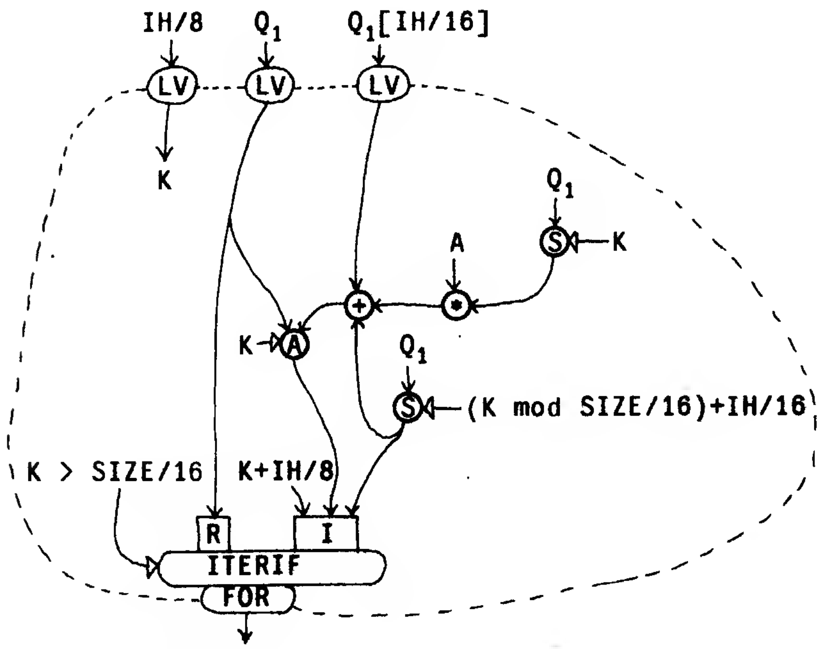
To properly compile this, the identity

$$\mathrm{mod}(16*K+1,\ SIZE) = 16*\mathrm{mod}(K,\ \frac{SIZE}{16})+1$$

is used. (Yes, optimizing compilers will need a fair amount of virtuosity in dealing with modular arithmetic.)

The resulting graph, for IH$\geq$16, is

Fig. 4.13

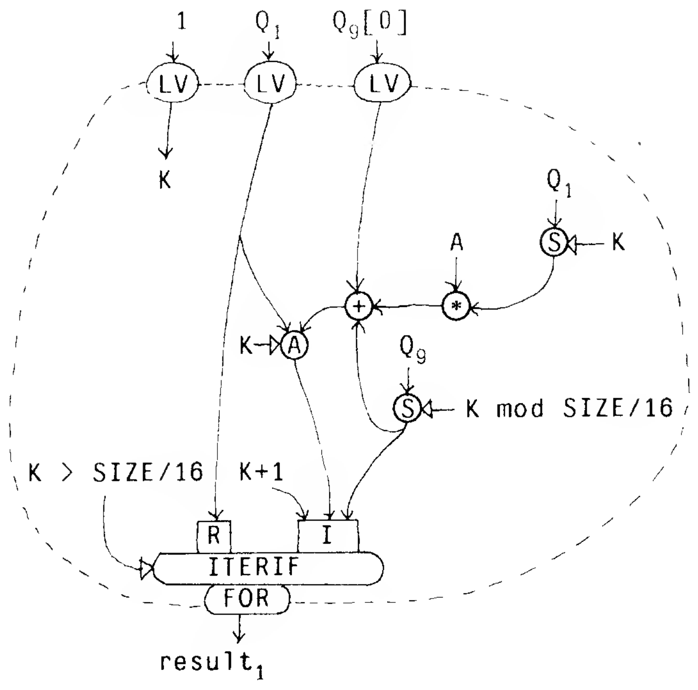


```
result₁
(other slices are just the
corresponding slices of Q)
```

In the remaining graphs we also use a few arithmetical tricks.

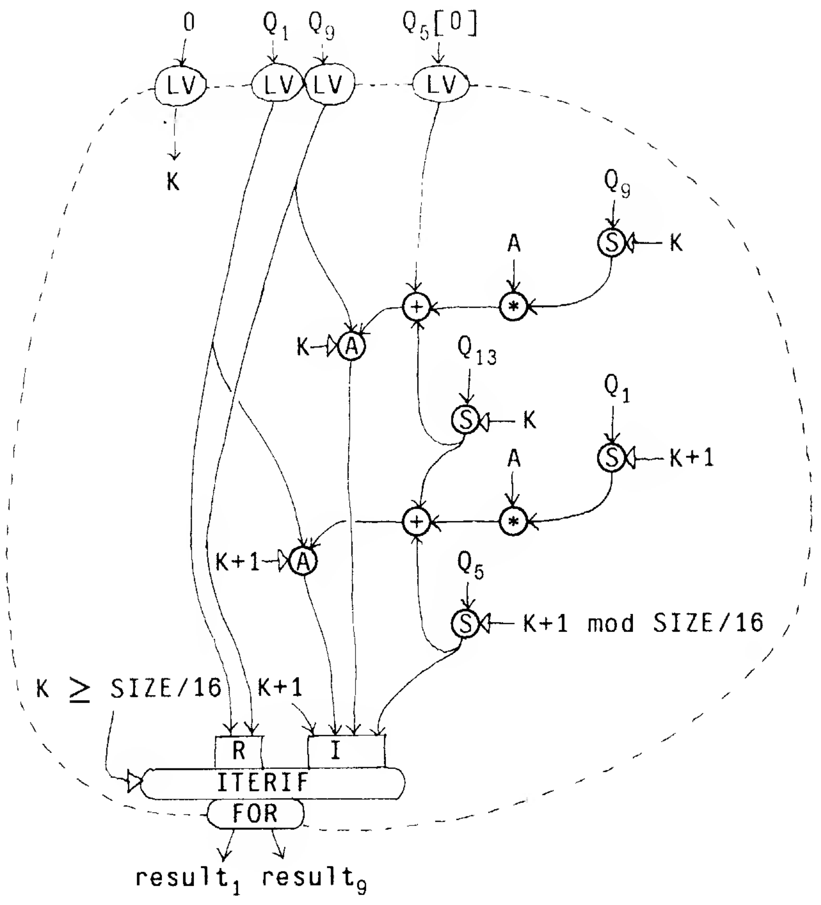
The graph for IH=-8 is

Fig. 4.14



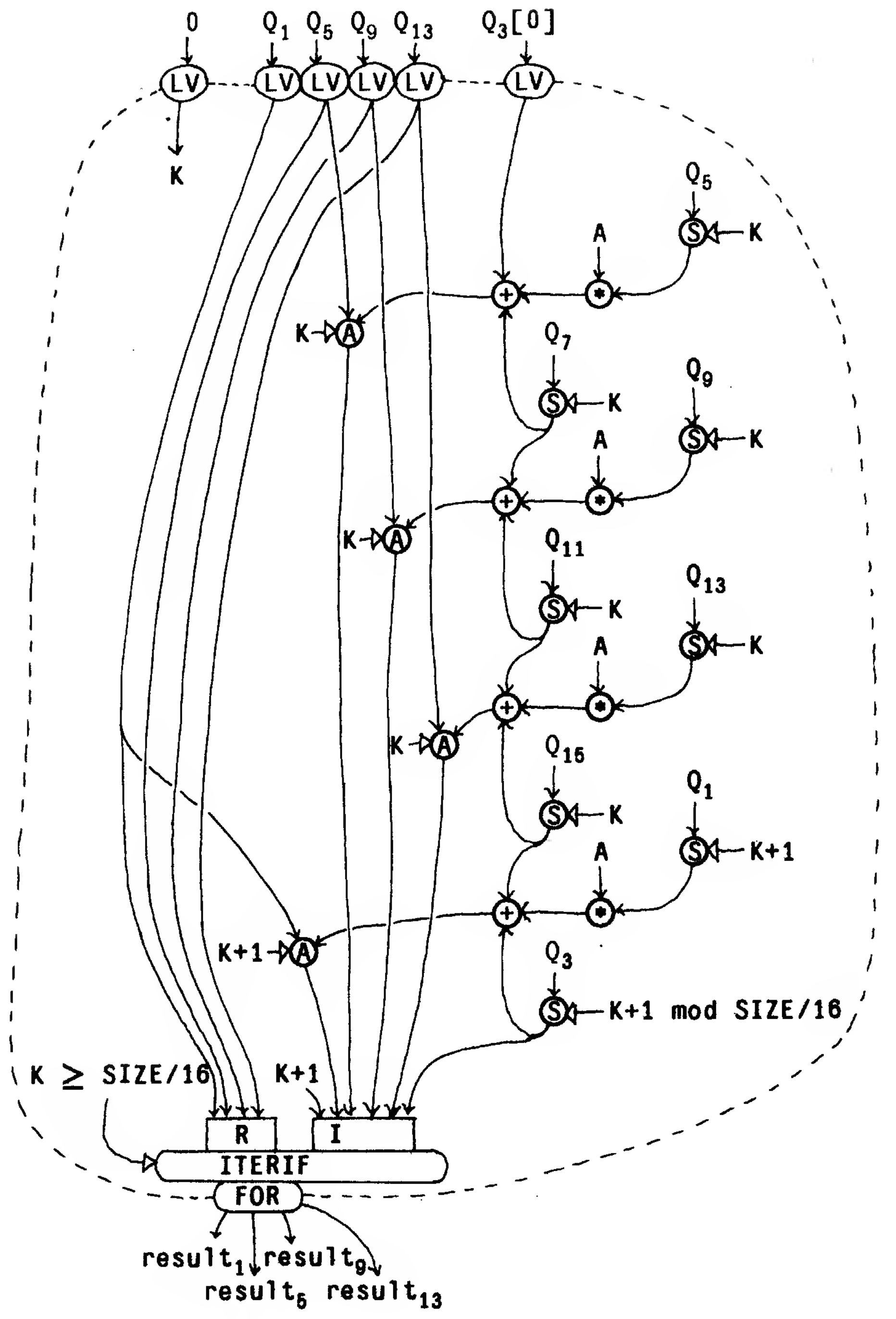
The graph for IH = 4 (unfolding = 2) is

Fig. 4.15



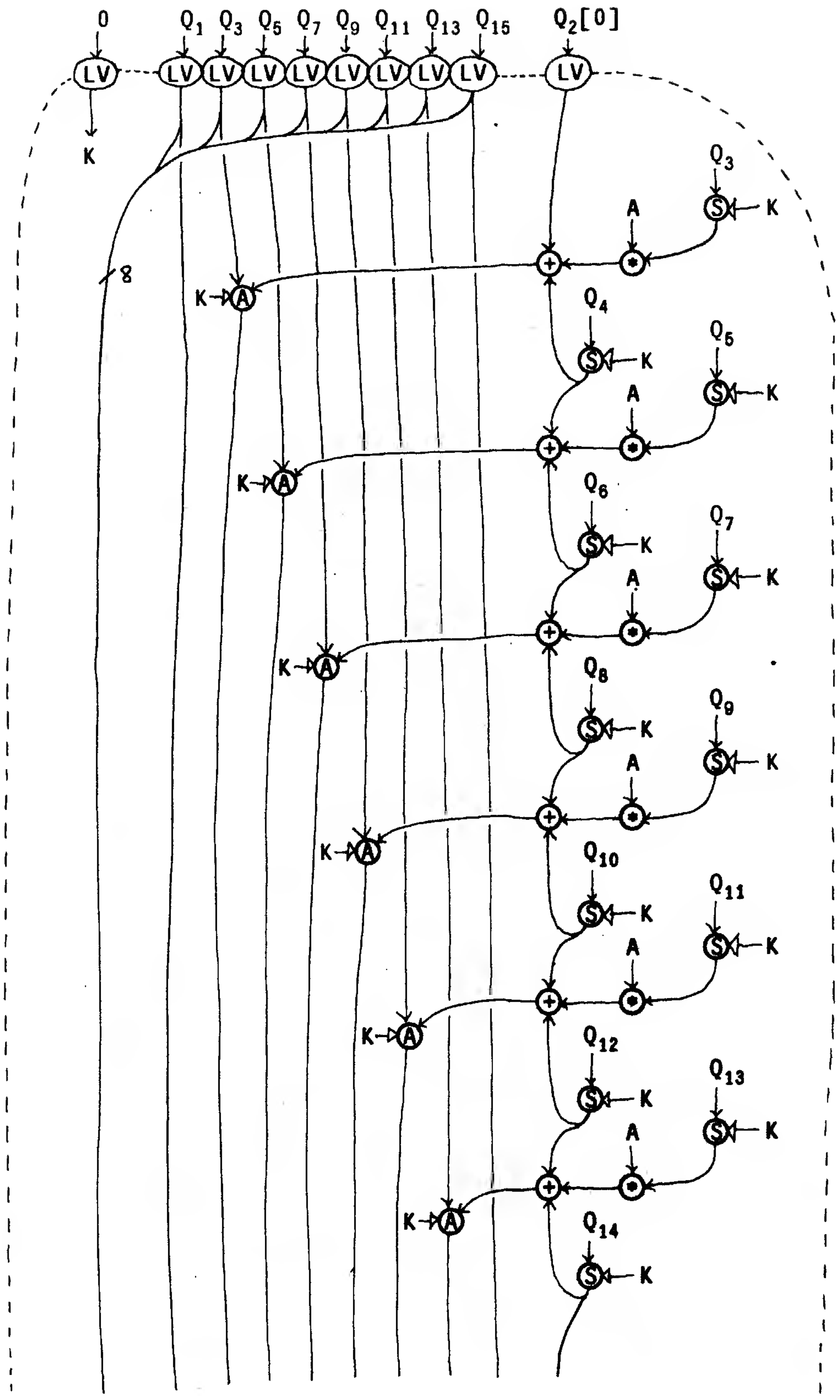
The graph for IH = 2 (unfolding = 4) is

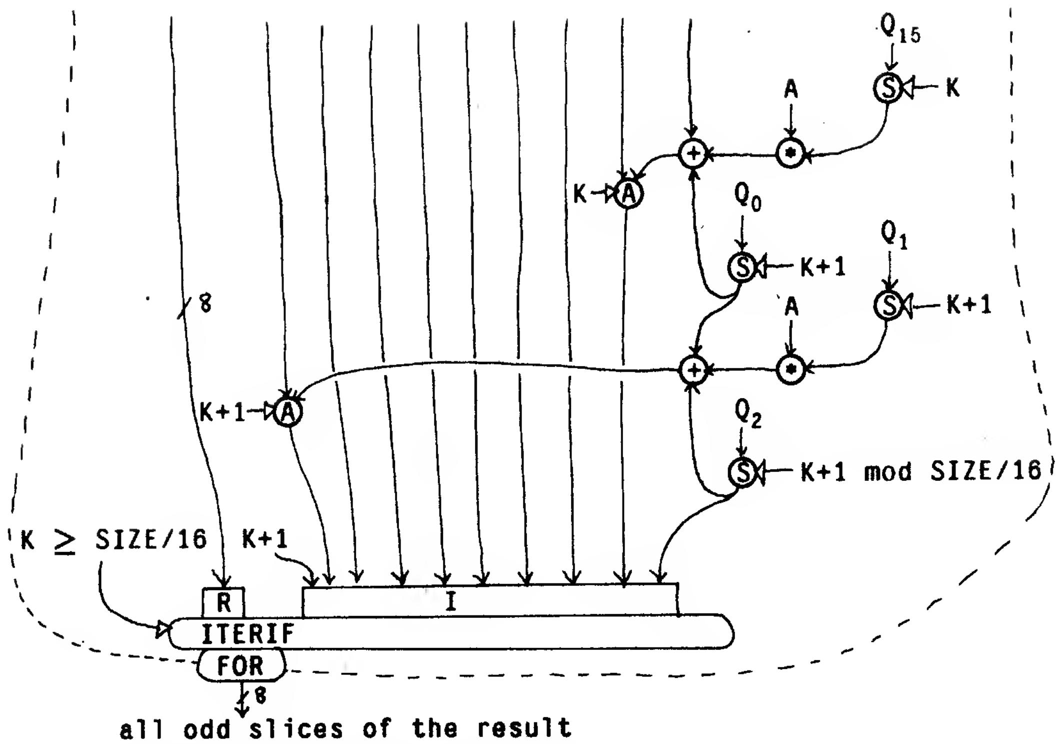
Fig. 4.16



The graph for IH = 1 (unfolding = 8) is

Fig. 4.17


0
Q_1 Q_3 Q_5 Q_7 Q_9 Q_{11} Q_{13} Q_{15}
$Q_2[0]$
LV
K
8
A
S
Q_3 Q_4 Q_5 Q_6 Q_7 Q_8 Q_9 Q_{10} Q_{11} Q_{12} Q_{13} Q_{14}



all odd slices of the result

To summarize the transformation of the inner loop of the cyclic reduction algorithm:

1. We assumed an optimizer that is quite skillful at performing transformations on integer computations, for such things as loop reparameterization, calculation of termination conditions, and removal of mod operators where the known range of the arguments makes them trivial.

2. We made heavy use of the fact that we knew the value of the parameter IH, at least for the values less than 16. For the other values, we used the fact that it was a multiple of 16 so that array references could be generated without knowing its exact value. How we use the fact that IH is a power of two, and how we must deduce that from the algorithm's outer loop, further illustrate the need for a good integer optimizer.

3. We made use of the fact that SIZE is a power of two and at least 16, though nothing more need be known about it.

4.10 The Periodic Cyclic Reduction Algorithm -- Reduction Part, Outer Loop

We now examine the outer loop, and see how the loops interact.

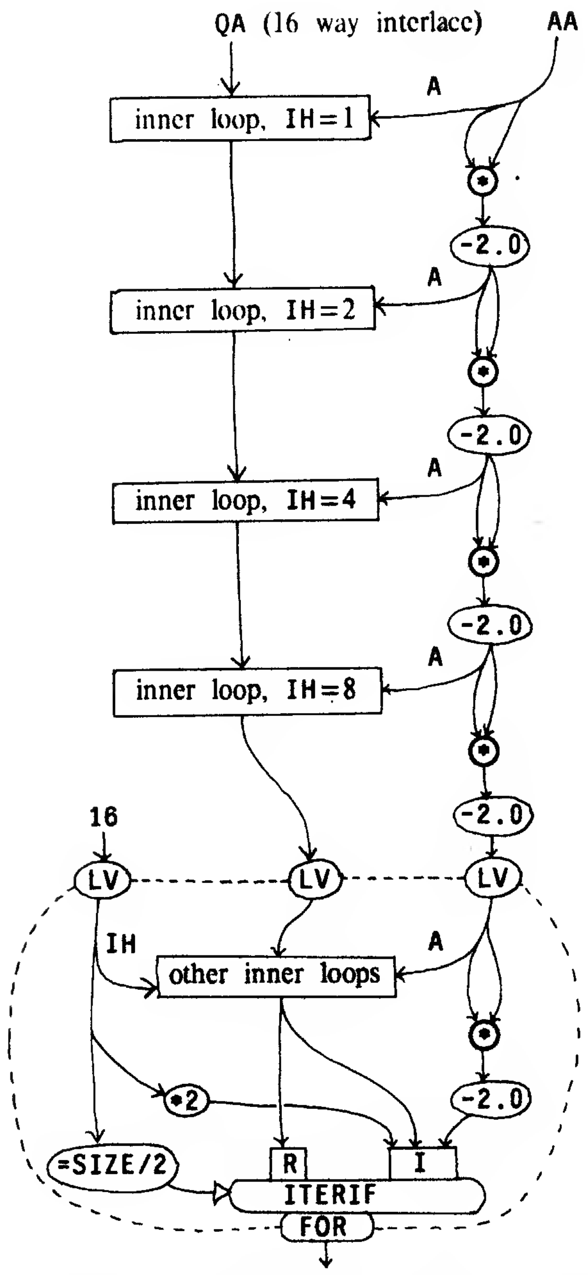
The simplified outer loop is:

```
for IH := 1 ;                     % IH runs from 1 through SIZE/2
    A := 2.0-C*FAC ;
    Q := QA ;                     % QA is the actual incoming array
do  let NEWQ := <inner loop, depends on Q, A, and IH> ;
    in  if IH=SIZE/2 then NEWQ
        else iter IH, A, Q := IH*2, A*A-2.0, NEWQ enditer
        endif
    endlet
endfor
```

The usual translation of a loop would consist of one copy of the body, with appropriate control operators to cause the body to be used repeatedly. In this case, the inner loops for IH=1, 2, 4, and 8 must be different code. This can be handled by performing an initial expansion of 4 on the outer loop. Whenever an inner loop has different structure on certain cycles of the outer loop, and those cycles occur at the beginning or end, those loops can be pulled out of the outer loop by making and initial or final expansion.

In this case, the expanded outer loop would look like

Fig. 4.18



4.11 The Periodic Cyclic Reduction Algorithm -- Substitution Part

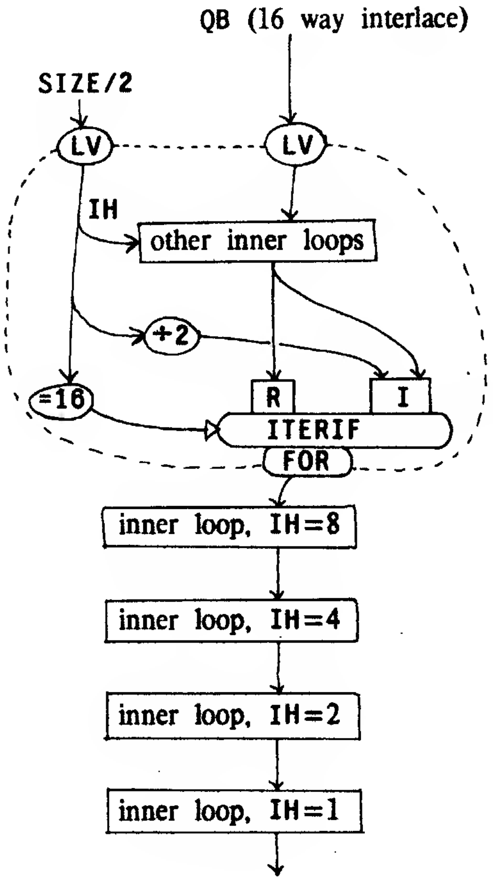
The "substitution" part of the cyclic reduction algorithm is very similar to the "reduction" part. The principal difference is that the inner loop is called with the values of IH in reverse order, from SIZE/2 down to 1. The inner loop is very similar to the corresponding loop for the "reduction" part, and is analyzed in the same way. It requires one loop for IH$\geq$16, and separate loops for IH equal to 8, 4, 2, and 1.

The simplified outer loop is

```
for IH := SIZE/2 ;                    % IH runs from SIZE/2 to 1
    Q := QB ;                         % QB is the result of the reduction
do  let NEWQ := <inner loop, using Q and IH> ;
    in  if IH=1 then NEWQ
        else iter IH, Q := IH/2, NEWQ enditer
        endif
    endlet
endfor
```

Because the special values of IH -- 8, 4, 2, and 1, occur at the end of the outer loop, a *final* expansion is used. The resultant graph is

Fig. 4.19



The "cyclic reduction algorithm" can be used to solve other types of difference equations. Another version of it, to solve tridiagonal linear systems, will be presented in full in Chapter 13. The version shown here was selected because only one array is used in the critical parts, and because the periodic boundary condition (which gave rise to the mod operator) shows an interesting example of the types of optimization that must be performed on integer expressions.

5. UNFOLDING AND INTERLACE IN MULTIPLE DIMENSIONS

In this chapter the results of the previous chapter will be generalized to the cases of nested iteration loops and multidimensional arrays.

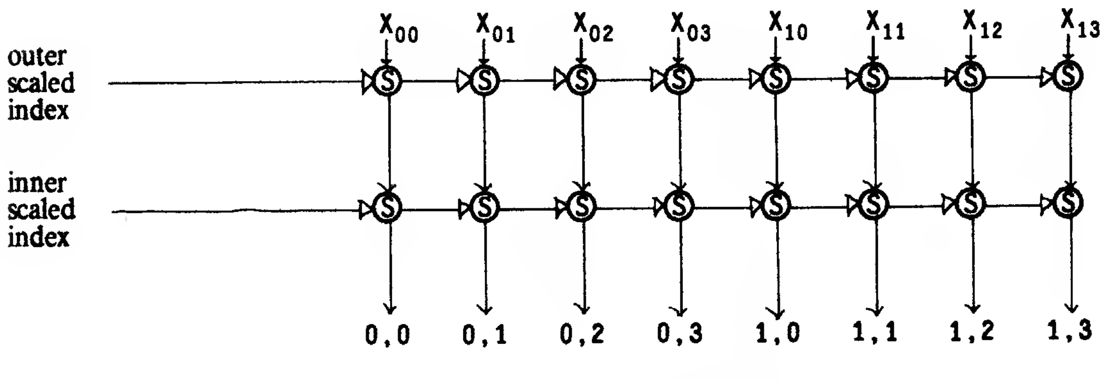
Multidimensional arrays can be interlaced at each level. If a two-dimensional array `X` has "2*4 interlace", it has a 2-way interlace at the top level, and each element of each top level slice is a one-dimensional array with a 4-way interlace. The total array thus has 8 slices, denoted X_{ij} with $0 \leq i < 2$ and $0 \leq j < 4$, such that

`X[2P+i][4Q+j]` = X_{ij}`[P][Q]` where `P` and `Q` are arbitrary expressions
(in the source program) (in the implementation)

Multidimensional arrays and nested loops work very well together in the simple cases. As one might expect, the nicest case is the one in which the loop nesting and array indexing match each other, the interlace is equal to the unfolding at each level, and the reference interval is equal to one at each level.

Here is the graph for references to `X[I][J]` in a nested loop with `I` as the outer loop index, `J` as the inner loop index, and 2*4 unfolding and interlace:

Fig. 5.1



loop instances

5.1 Relationship Among Unfolding, Reference Interval, and Interlace

If, at some level, the unfolding and interlace are not equal, or the reference interval is not equal to one, the same thing happens at that level as in the one-dimensional case.

$\boxed{\texttt{reference interval * unfolding < interlace:}}$

Permute operators will be needed, with $\dfrac{\texttt{interlace}}{\texttt{reference interval*unfolding}}$ inputs.

$\boxed{\texttt{reference interval * unfolding > interlace:}}$

Each array slice that is used goes to $\dfrac{\texttt{unfolding}}{\left\lceil \dfrac{\texttt{interlace}}{\texttt{reference interval}} \right\rceil}$ array operation nodes.

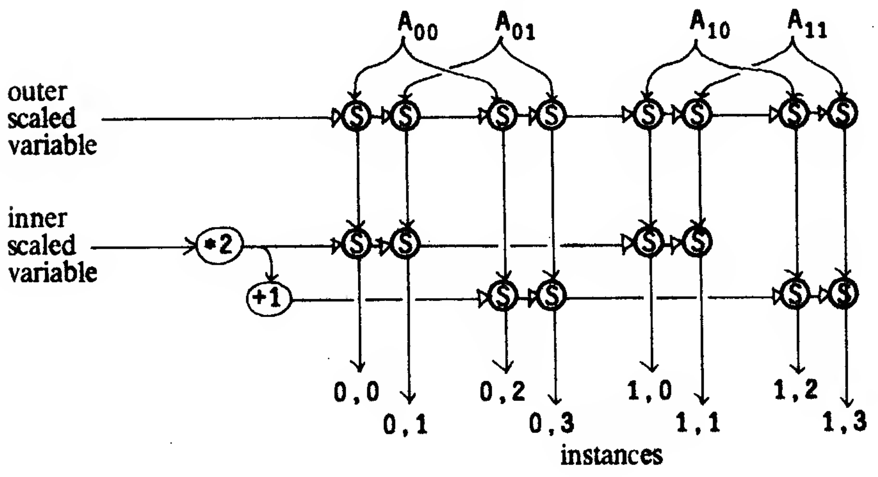
Both situations are nonoptimal; the first one is more serious.

These effects can be illustrated through the preceding example of a nested loop with 2*4 unfolding, by setting the interlace to something other than 2*4.

Suppose the array A has 2*2 interlace. Then the select's from it look like

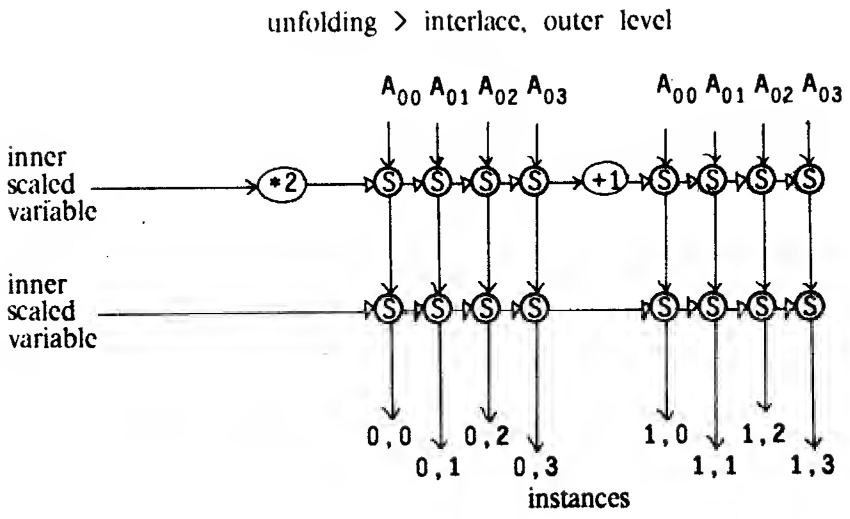
Fig. 5.2

unfolding > interlace, inner level



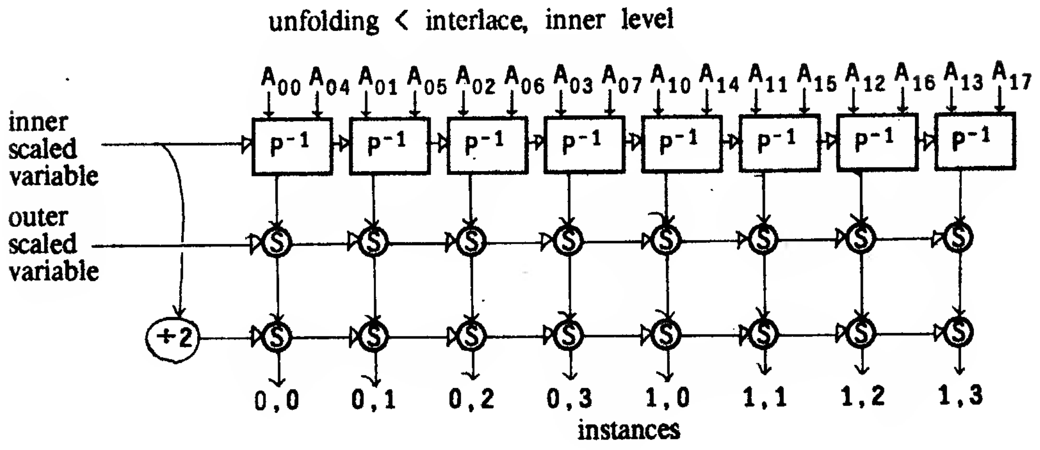
Suppose the array A has 1*4 interlace. Then the select's from it look like

Fig. 5.3


unfolding > interlace, outer level
A_{00} A_{01} A_{02} A_{03}
A_{00} A_{01} A_{02} A_{03}
inner scaled variable
*2
+1
inner scaled variable
0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3
instances

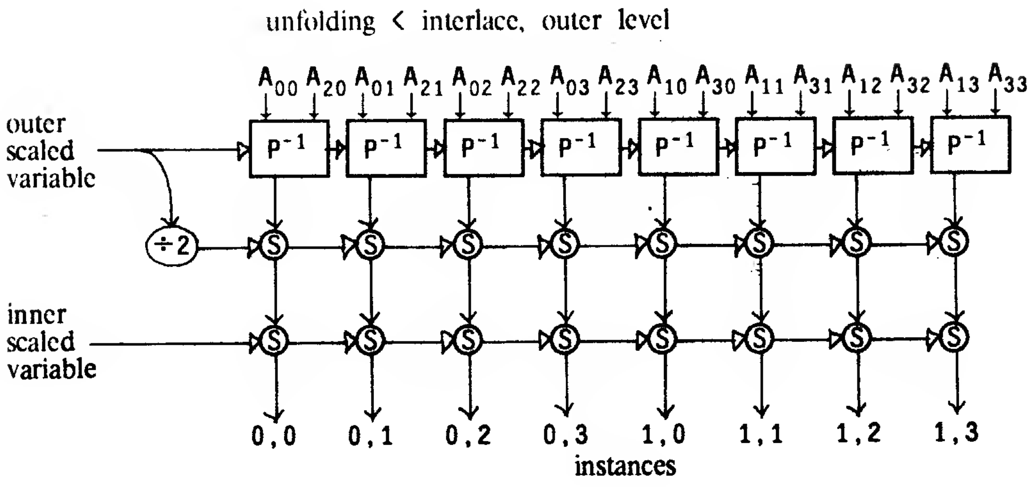
Suppose the array A has 2*8 interlace. Then the select's from it look like

Fig. 5.4


unfolding < interlace, inner level
A_{00} A_{04} A_{01} A_{05} A_{02} A_{06} A_{03} A_{07} A_{10} A_{14} A_{11} A_{15} A_{12} A_{16} A_{13} A_{17}
inner scaled variable
outer scaled variable
p^{-1}
+2
0,0
0,1
0,2
0,3
1,0
1,1
1,2
1,3
instances

Suppose the array A has 4*4 interlace. Then the select's from it look like

Fig. 5.5


unfolding < interlace, outer level
A00 A20 A01 A21 A02 A22 A03 A23 A10 A30 A11 A31 A12 A32 A13 A33
outer scaled variable
P-1
÷2
S
inner scaled variable
0,0 0,1 0,2 0,3 1,0 1,1 1,2 1,3
instances

The preceding data flow graphs are just models of the program organization at a certain conceptual level. When prepared for execution, further transformations must be made: The **permute** operators are moved downstream so that they handle only scalar data; consecutive **select** operators are combined as multidimensional array slices are flattened, etc.

5.2 Unfolding and Interlace That do Not Nest Similarly

In the preceding, we have considered only cases in which the depth of loop nesting was the same as the dimensionality of the array, and furthermore that the array indices were in exact correspondence with the loop indices. That is, the outermost array reference depended only on the outermost loop index, and so on. In fact, we can have much less regular arrangements, such as reversal of the order of indices:

```
for I := ....
    ....
    for J := ....
        ....
        A[J][I]
    endfor
endfor
```

or an array reference depending on two loop indices:

```
A[8*J+I]
```

or none at all:

```
A[7]
```

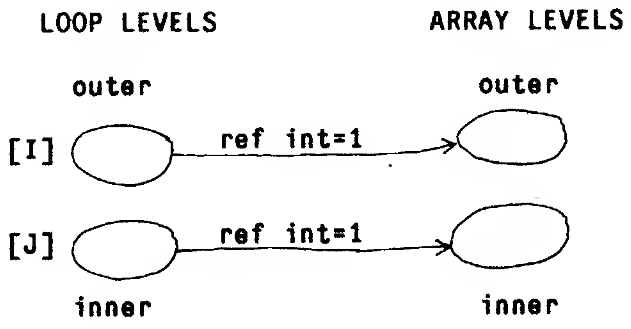
or any combination:

```
A[J][7][I+2*J][I][4*J]
```

These variations have no significant effect on the actual array select operations because, once the array slices are flattened, the select operation involves an array index which is some affine function of the loop indices. It is fairly straightforward to determine what that affine function is.

The important task is to examine the relationship among reference interval, interlace, unfolding, and the resultant code for every dependence of an array operation on a loop index. In the previous examples, we assumed a strict correspondence between loop levels and array levels, with reference interval of one in each case. This could be represented graphically by a *nesting diagram* as follows:

Fig. 5.6

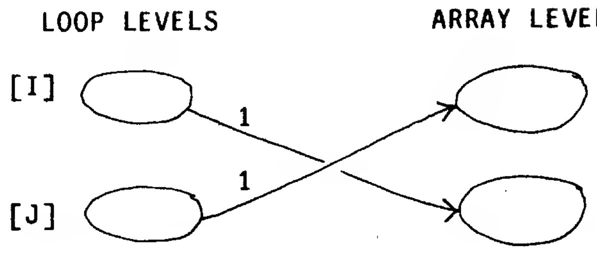


An arrow is drawn whenever there is a dependence, and each arrow has a reference interval associated with it.

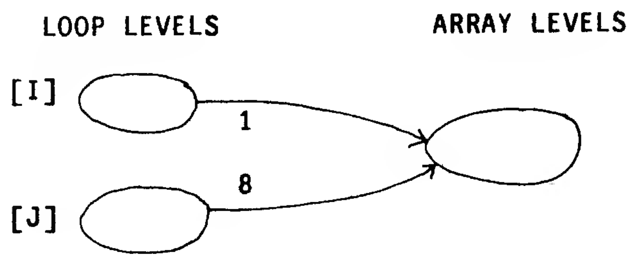
The examples illustrated earlier would have the following nesting diagrams:

Fig. 5.7

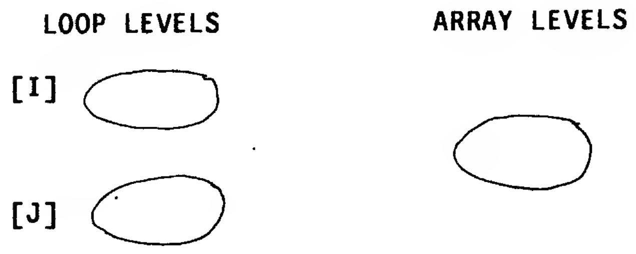
A[J][I]



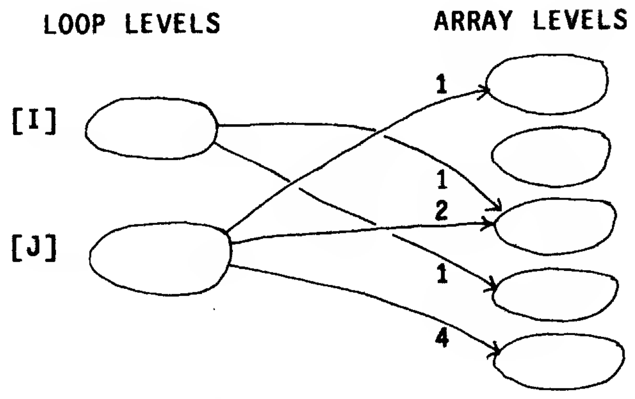
A[8*J+I]



A[7]

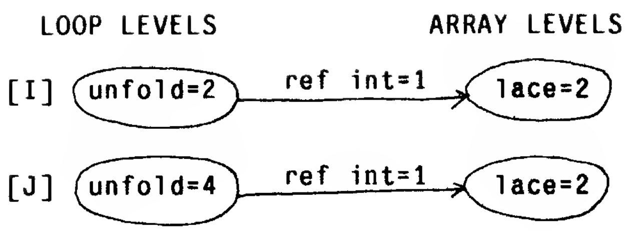


A[J][7][I+2*J][I][4*J]



The loop unfolding at each level, and the array interlace at each level, are properties of the loops and arrays themselves, not of the references, so they can be written inside the circles. For the example of 2*4 unfolding, 2*2 interlace, and strict nesting, the diagram is

Fig. 5.8



We know that the property of a reference that determines what kind of array operators are required to implement it is the comparison between the interlace and the product of the unfolding and the reference interval. In the general case, this comparison is made for each arrow -- the interlace at the right end of the arrow is compared with the product of the arrow's reference interval and the unfolding at its left end. In this example, there is a mismatch in the lower arrow -- the interlace is too small -- so array slices are used in multiple places but **permute** operators are not required.

If the nesting diagram is complicated, there may be no choice of unfoldings and interlaces that leads to a good match everywhere.

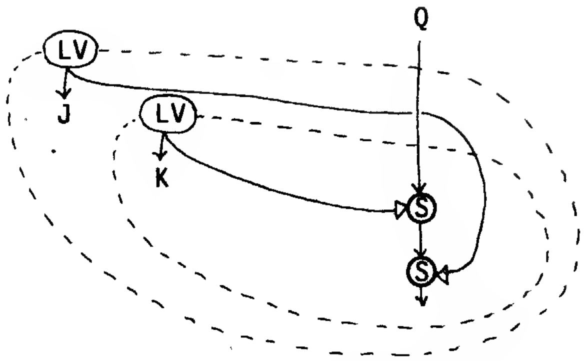
If an array node in the nesting diagram has no arrows going into it, that reference has a constant index (at least one that does not depend on any of the loop variables), so the situation is trivial. If it has exactly one arrow going into it, the method developed so far can deal with it, even if the arrows cross each other.

An example of crossed but otherwise simple arrows arises in the 2-dimensional cyclic reduction algorithm. Part of the code is

```
forall J in [....]
    ....
    forall K in [....]
        ....
        Q[K][J]
    endall
endall
```

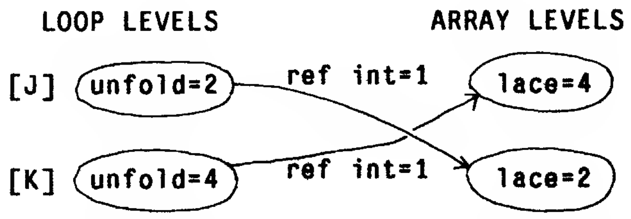
The graph fragment is

Fig. 5.9



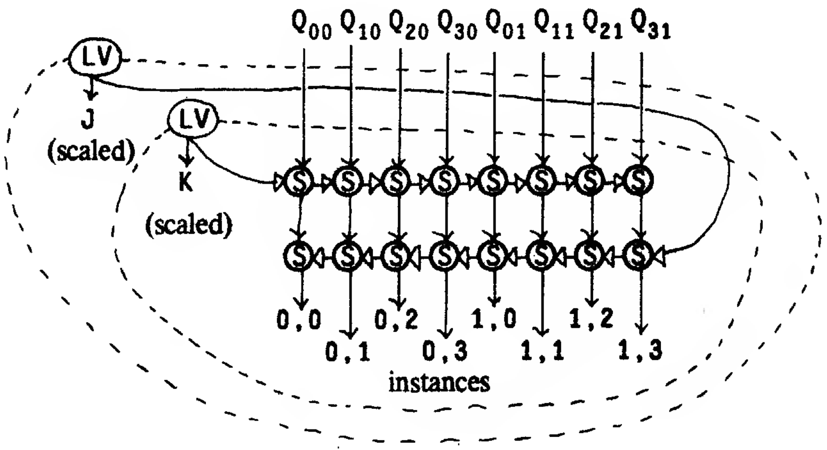
If the array Q has 2*4 interlace, we could choose 4*2 interlace:

Fig. 5.10



Both arrows match perfectly. The array operations in the unfolded loops are:

Fig. 5.11



5.3 Array References Depending on More Than One Variable

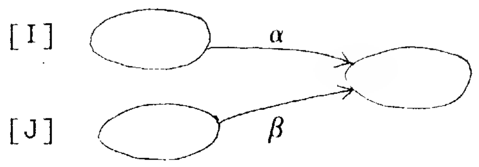
If two or more arrows point to the same array node in the nesting diagram, the array reference depends on two or more loop variables.

For example

```
A[α*I + β*J + γ]
```

gives rise to

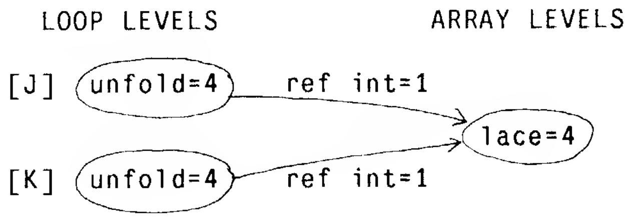
Fig. 5.12



The graph resulting from loop expansion and array interlace expansion is generally similar to the cases discussed previously: In each loop instance, some array slice (perhaps selected by a permuter controlled by the scaled loop variables) goes to an array operator whose index argument is a function of the scaled loop variables.

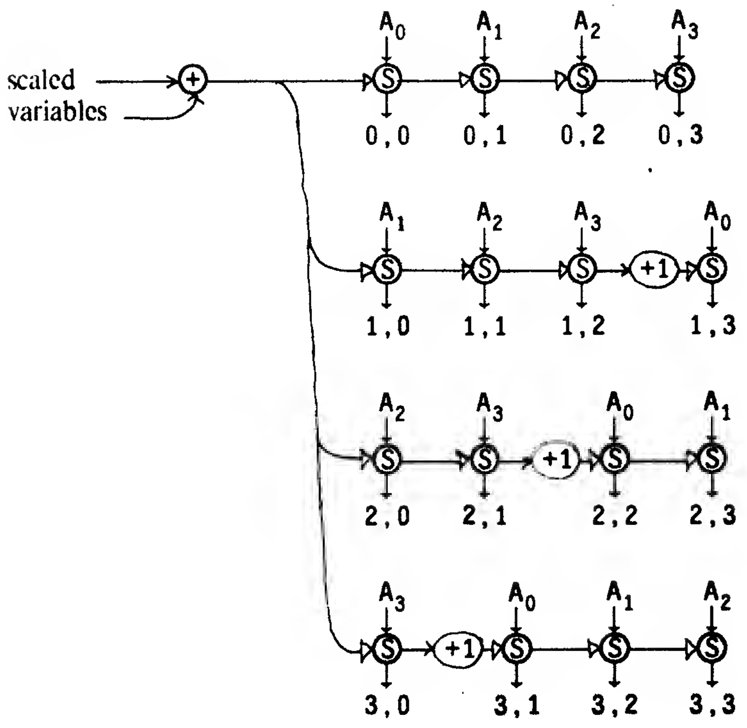
Example of reference "A[I+J]" with 4*4 unfolding and 4 way interlace

Fig. 5.13



There are 16 loop expansions. The select operations look like

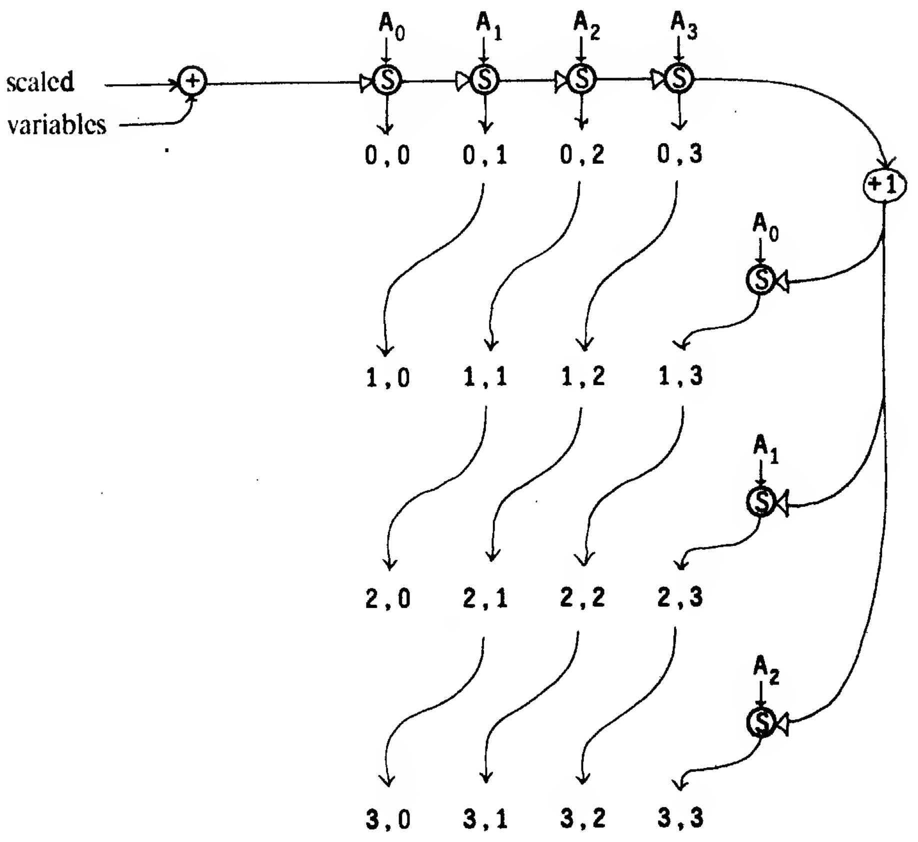
Fig. 5.14



In the one-dimensional case, there was a rule that, if the reference interval times the unfolding is greater than the interlace, each array slice goes to several array operation nodes. That rule does not apply when the reference depends on more than one loop variable. Each slice typically goes to many operations, regardless of the unfolding and interlace, because there are typically many values of the variables I and J that have the same sum. Of course a code generator will optimize out redundant references to the same array slice.

In the example above, the optimized array selections might look like:

Fig. 5.15



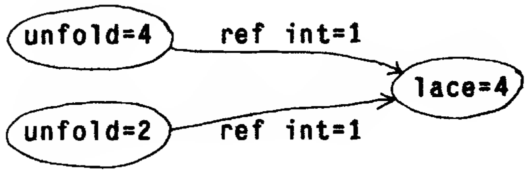
To find out whether **permute** operators are required, note that we have many (`reference interval * unfolding`) products. The *smallest* of them is the important parameter. If this is less than the array interlace, a **permute** operator is needed, and that **permute** must choose from among

$$\frac{\texttt{interlace}}{\texttt{min \{reference interval*unfolding\}}}$$

inputs.

Example:

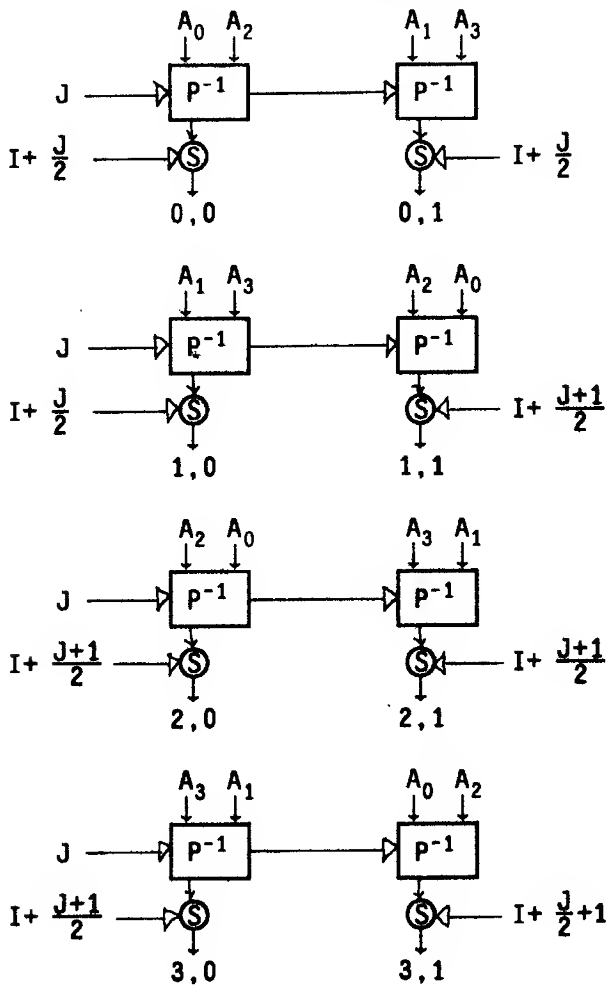
Fig. 5.16



This needs a 2-way permuter on the inner index. All division operators are assumed to truncate nonintegral results downward:

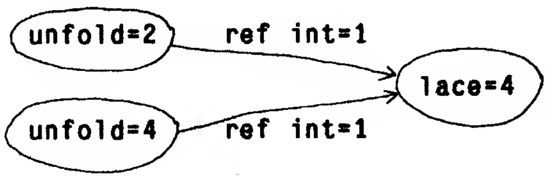
Fig. 5.17

Scaled outer index = I Scaled inner index = J



Another example:

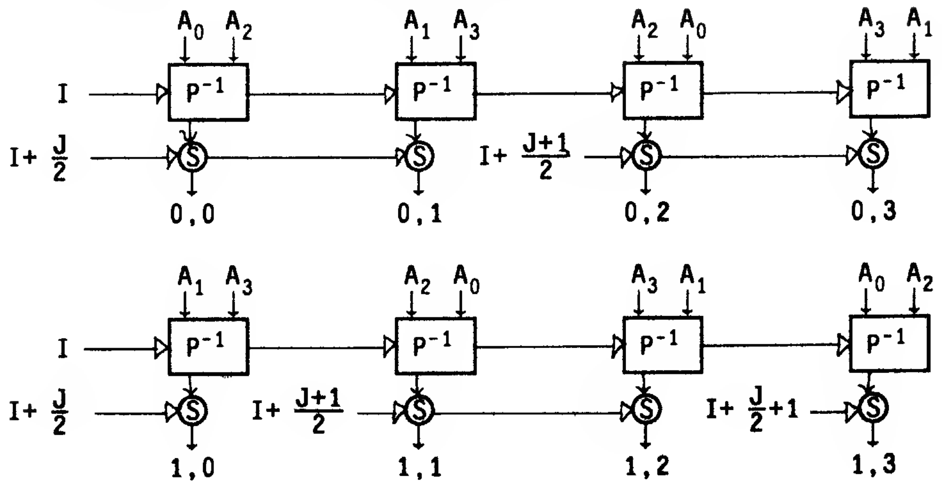
Fig. 5.18



This needs a 2-way permuter on the outer index:

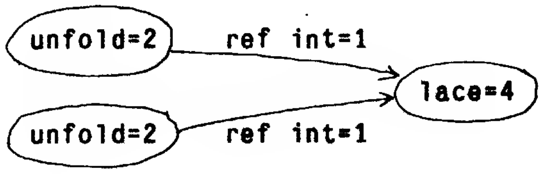
Fig. 5.19

Scaled outer index = I   Scaled inner index = J



Example with `unfolding < interlace` on two variables:

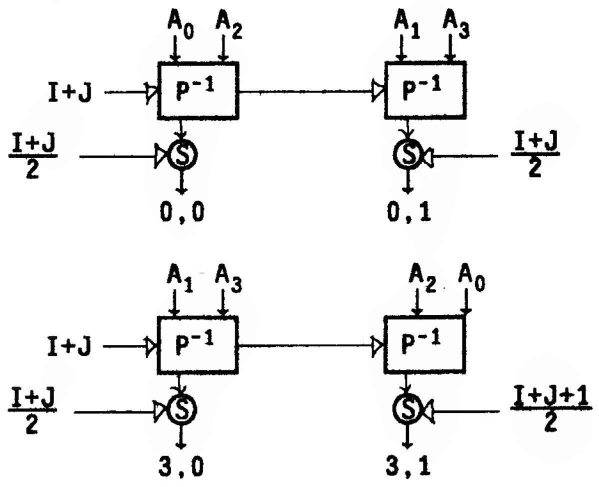
Fig. 5.20



This needs a 2-way permuter on a function of both indices:

Fig. 5.21

Scaled outer index = I   Scaled inner index = J



If all of the `(reference interval * unfolding)` products are greater than the interlace, the structure becomes different, but permuters are not required.

Fig. 5.22

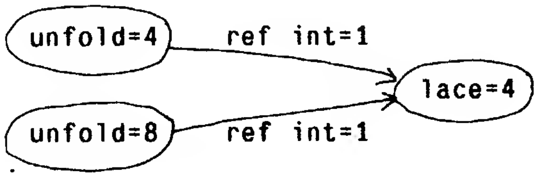
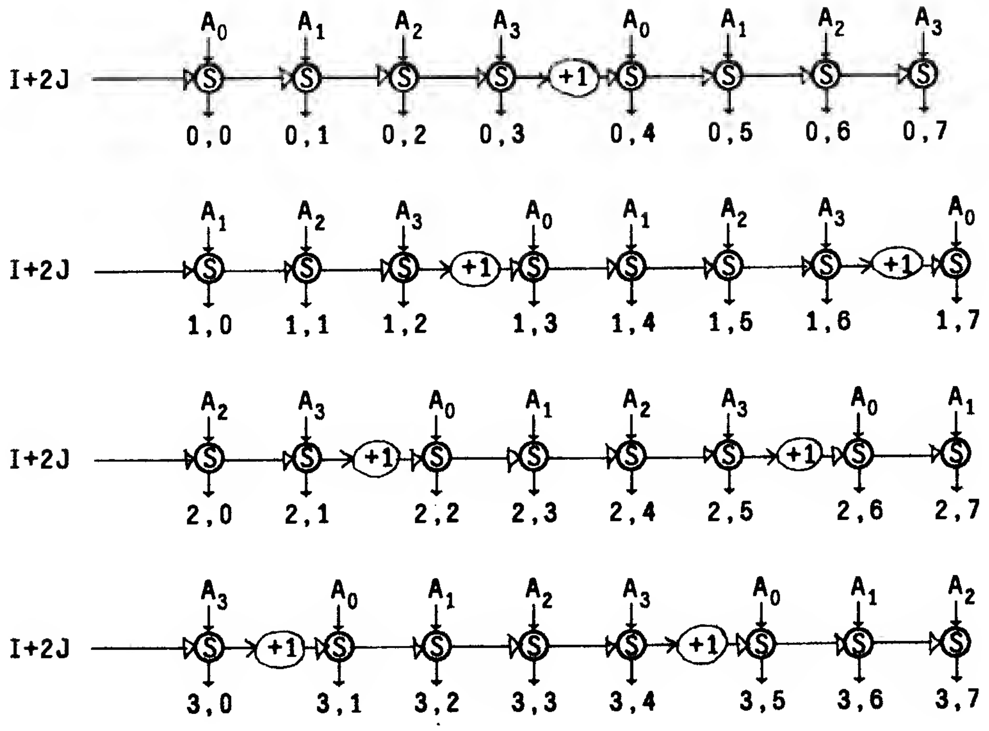


Fig. 5.23

Scaled outer index = I   Scaled inner index = J



Here is an example in which one of the (`reference interval * unfolding`) products is greater than the interlace and one is smaller. The smaller one creates the need for a permuter.

Fig. 5.24

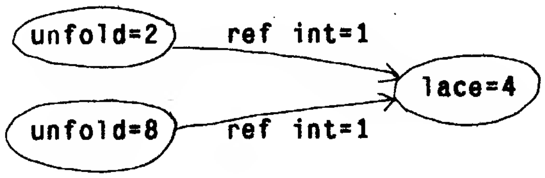
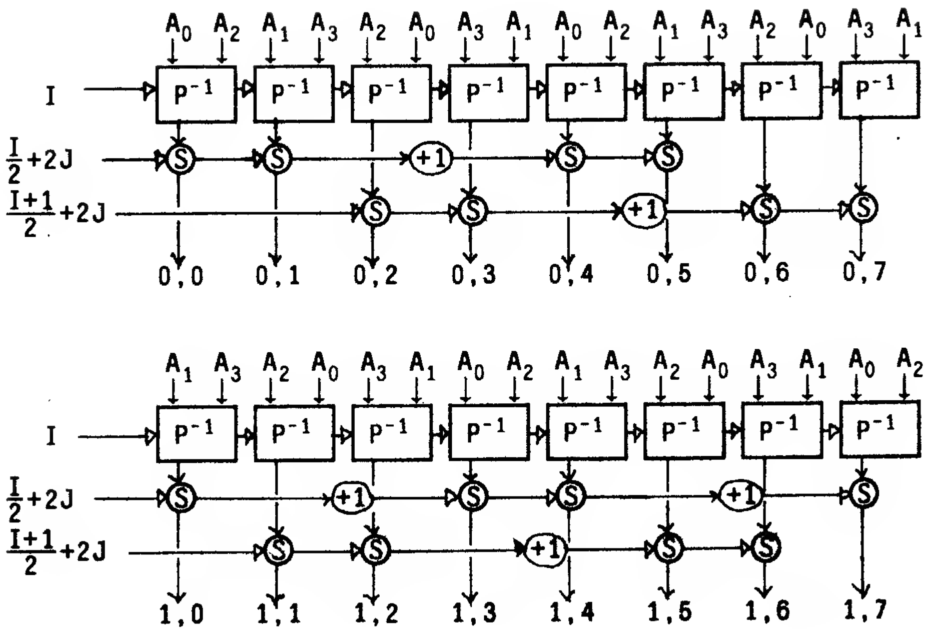


Fig. 5.25

Scaled outer index = I Scaled inner index = J



6. COMBINING COMMON SUBEXPRESSIONS

Finding common subexpressions within an expression or program fragment is a very well known optimization for conventional compilers. It is of extreme importance, particularly with regard to array references, in an applicative supercomputer.

A "common subexpression" is something like "`X-Y`" in "`(X-Y)*(X-Y)`". Transforming it to "`let T := X-Y in T*T`" saves a subtraction and perhaps a few memory cycles on a conventional computer.

This optimization actually has limited usefulness in conventional situations, because common subexpressions of significant size are extremely rare. A human programmer is likely to change

```
    Y := (2.45*X**2 + 3.0) / (2.45*X**2 + 3.0 + X)
```

to

```
    T := 2.45*X**2 + 3.0
    Y := T/(T+X)
```

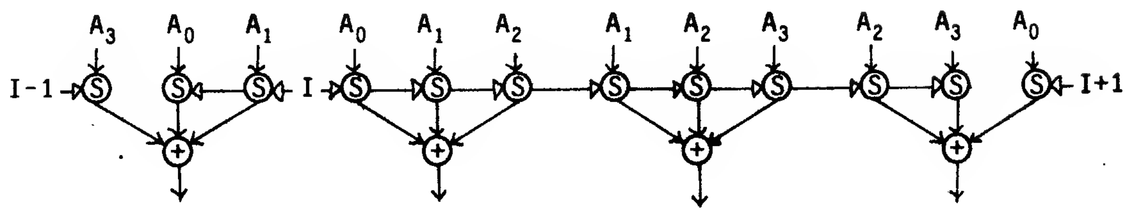
before a compiler ever sees it, if only to save typing.

In a data flow system with loop unfolding, however, optimization of common subexpressions becomes important because expressions in different loop expansions can be identical even though they looked different in the source program. That is, inter-cycle optimizations can be performed. A simple example of this is

```
    forall I in [LO, HI]
    .......
    ....  A[I-1] + A[I] + A[I+1] ....
    .......
```

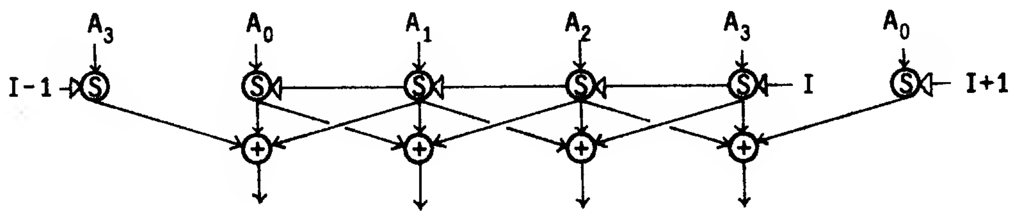
Assuming unfolding and interlace of 4, with rescaling, the four expansions are

Fig. 6.1



These can clearly be combined:

Fig. 6.2



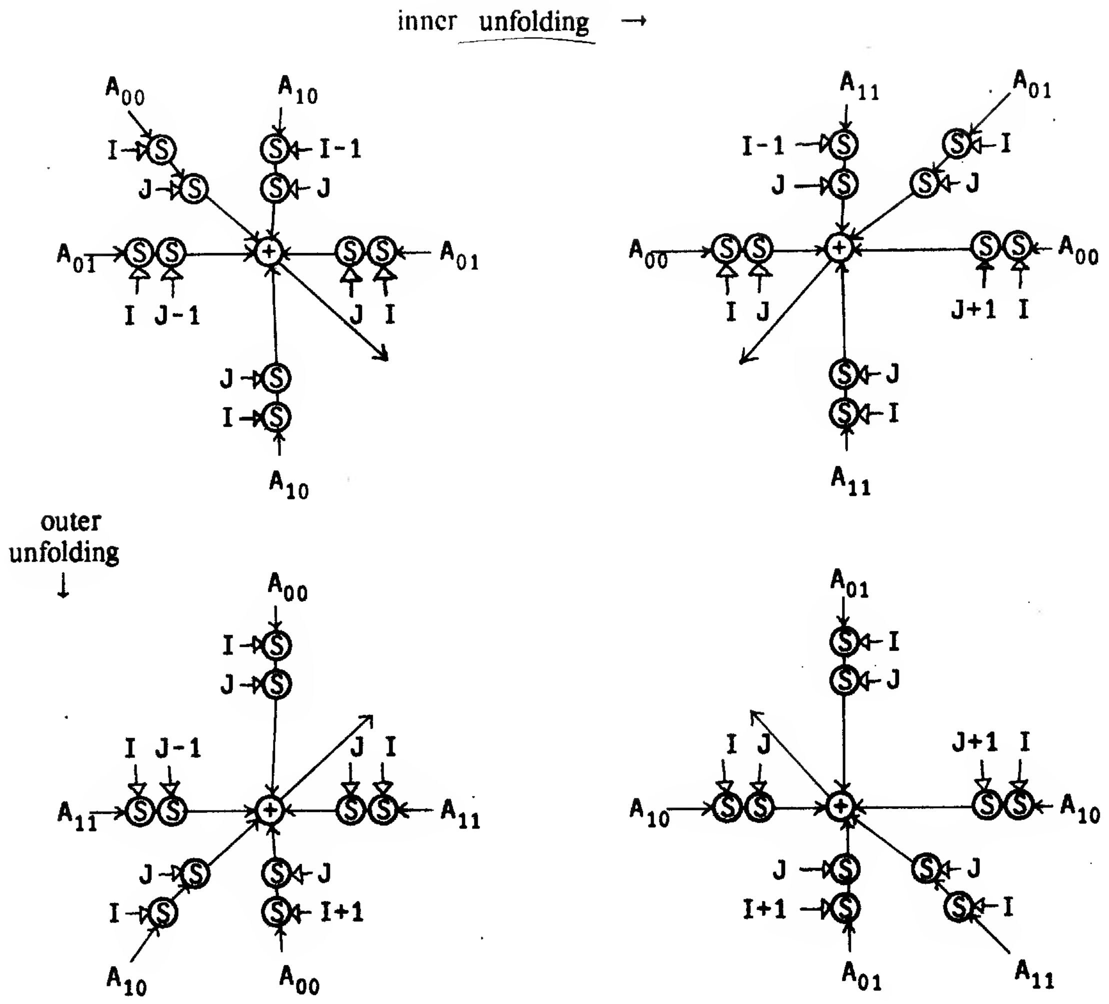
This is approaching the minimum possible number of array fetches. Except for the "boundary conditions" (which will be disposed of in the next chapter) each array element is fetched only once. Note that the "nearest neighbor" nature of the computation becomes graphically apparent in the above figure. We are heading in the direction of an "ideal" representation of the algorithm.

The same benefits can be realized in higher dimensions. An example in two dimensions is useful. The algorithm is the same "sum of self and adjacent points" problem as above.

```
forall I in [ILO, IHI]
    .......
    forall J in [JLO, JHI]
    .......
    ... A[I, J] + A[I+1, J] + A[I-1,J] + A[I, J-1] + A[I, J+1]
    .......
```

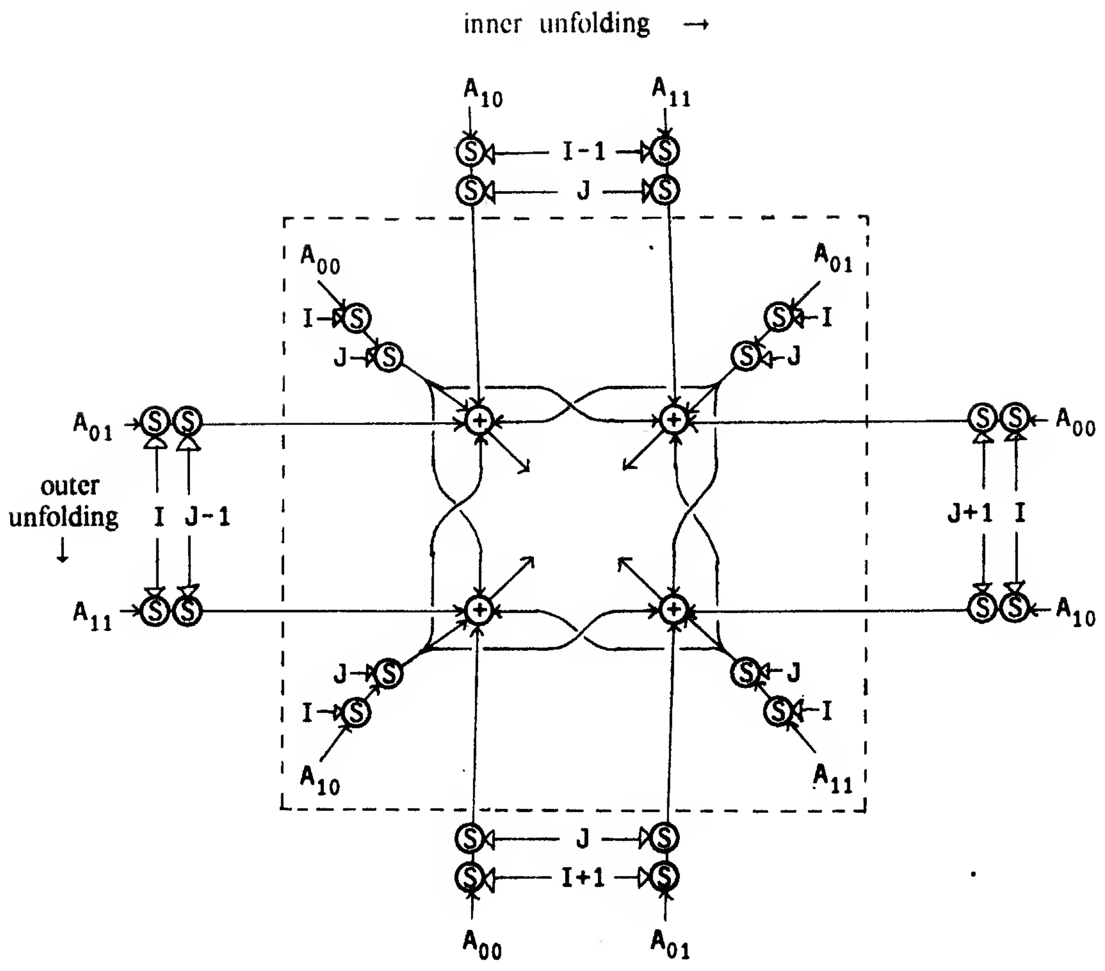
After a 2*2 unfolding and interlace, we have

Fig. 6.3



With removal of common subexpressions (which can take place before or after the interlace, and is independent of whether the interlace matches the unfolding well or badly) we have

Fig. 6.4

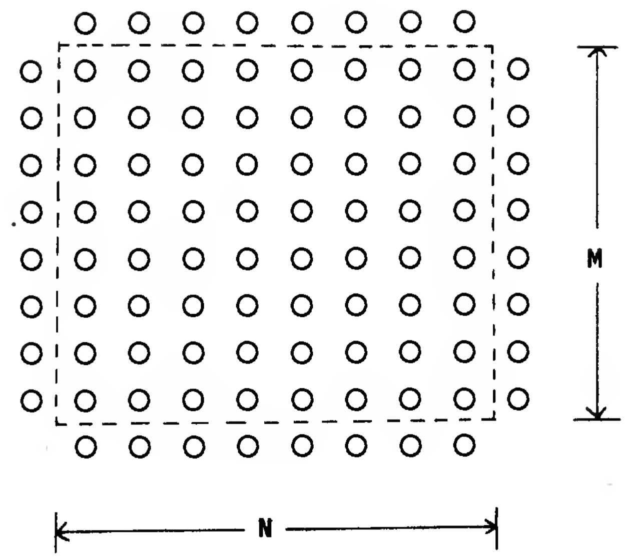


The part outside of the dotted line is the "boundary condition" computation. It gets proportionately smaller as the unfolding increases. The part inside the dotted line looks like what one would expect the data flow graph of the "sum of self and adjacent points" function to look like.

If there is an M*N unfolding, there are M·N array fetches[1] inside the dotted line and 2·M+2·N outside.

1. Remember from Section 3.11 that this will be turned into a single fetch after the array slices are flattened.

Fig. 6.5



We can now draw some conclusions about the optimal "shape" of nested loop unfoldings. (Shape was defined in Section 3.9.) If there is no communication among the loop unfoldings (no data dependencies) and they have no common subexpressions, there is hardly any reason to choose one shape over another. There will probably be a small difference in the number of operations required to control the loops, but this is a minor criterion and, in any case, needs to be evaluated in the context of many machine-dependent factors.

If there is communication among the cycles in an unfolding, say, common subexpressions as in the above example, then the shape should be more or less "square" (or cubical, or whatever). A well-known problem of elementary calculus is to find the shape that minimizes the perimeter for a given area. The solution is a square. In the above example, if we have space for 64 total unfoldings, an 8*8 shape would be best.

If the communication among the cycles in an unfolding is not isotropic, the result is different. If each grid point requires data from its neighbors two points away in the I direction and just one point away in the J direction:

```
A[I, J] + A[I+1, J] + A[I+2, J] + A[I-1, J] + A[I-2, J]
    + A[I, J+1] + A[I, J-1]
```

calculus once again provides the answer. The ideal shape is twice as long in the I direction as in the J direction. If there is no communication in the J direction, we may want to have all loop unfolding occur in the I direction.

The best shape for nested loop unfolding will also be strongly affected by the shape of the interlace of the arrays that are being used. The preceding discussion of optimal shape did not take into account the fact that array references are more expensive if the unfolding does not match the interlace. The interlace generally has to be chosen as a compromise to match as closely as possible the unfoldings of all of the loops that refer to the array. In practice, choosing all interlaces and unfoldings depends on a great many interacting influences.

7. AUXILIARY LOOP VARIABLES

In the preceding chapter we saw that common subexpressions can be combined among different source iteration cycles if those cycles are parts of the same unfolded cycle. We can also combine subexpressions from adjacent unfolded iteration cycles by saving the value in one cycle for use in the next. This is done by introducing extra iteration variables to carry the value.

Referring back to figure 6.2, in each unfolded iteration cycle the following six values were needed

$A_3[I-1]$ $A_0[I]$ $A_1[I]$ $A_2[I]$ $A_3[I]$ $A_0[I+1]$

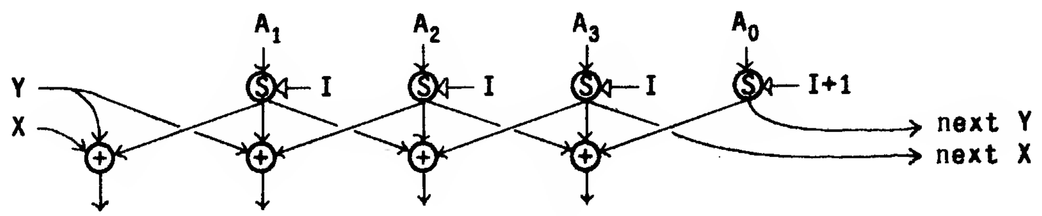
Four of them are inevitable, and two were required for "boundary conditions".

Note that, on each cycle, the references $A_3[I-1]$ and $A_0[I]$ are in common with two references from the previous cycle. On the previous cycle we calculated $A_3[I']$ and $A_0[I'+1]$, where I' was the index value on that cycle. Since $I' = I+1$, these are the values needed for the present cycle. Introduce variables X and Y, having the property that, during any cycle,

$$X = A_3[I-1]$$
$$Y = A_0[I]$$

Then the essential part of the program graph is

Fig. 7.1



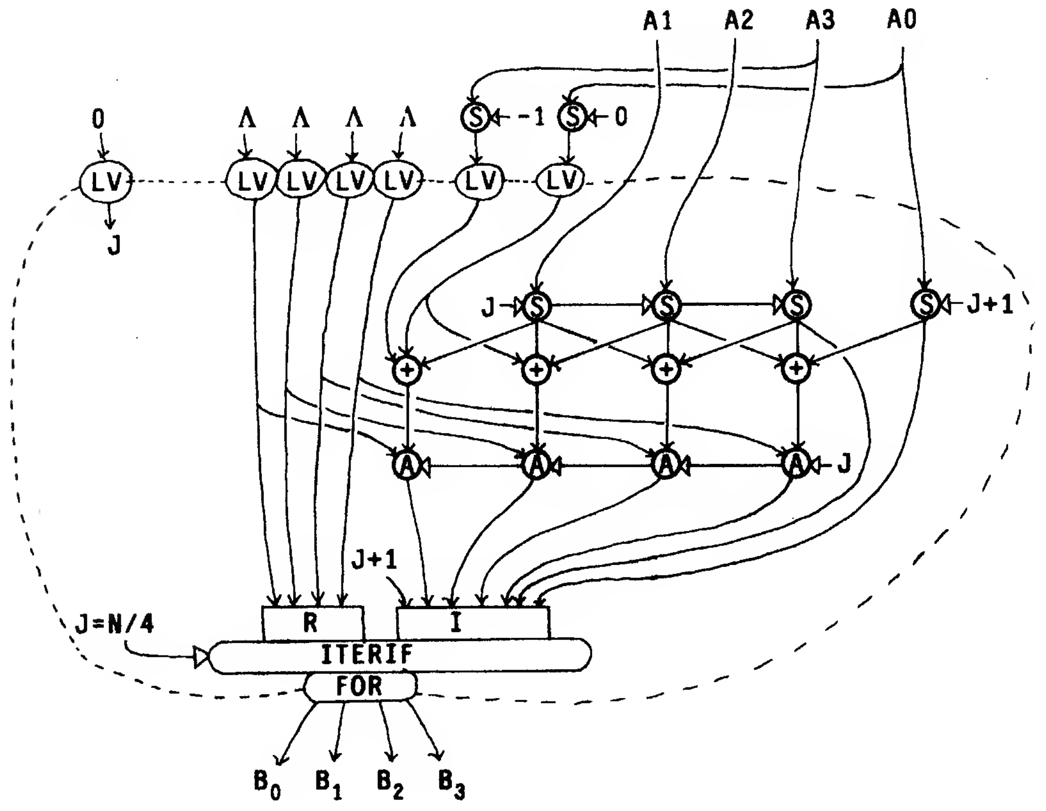
The initial values of X and Y must be set up appropriately, outside of the loop, so that the above equations will be true during the first cycle.

If the source program was

```
B := forall I in [0, N-1]
construct A[I] + A[I-1] + A[I+1]
endall
```

then, after 4-way unfolding and interlace, and introduction of auxiliary variables, we would have

Fig. 7.2



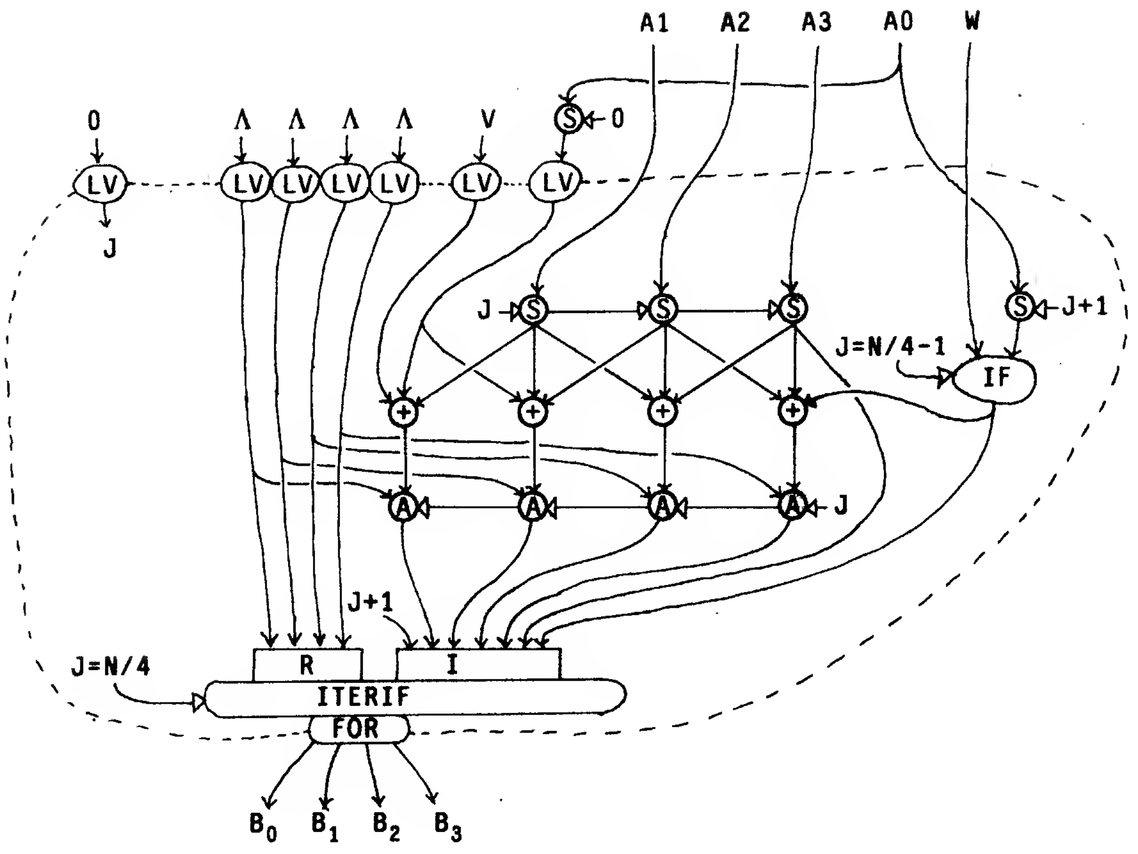
7.1 Boundary Conditions

Of course, the program might have been written with explicit boundary conditions:

```
B := forall I in [0, N-1]
construct
    A[I] +
    if I=0 then V else A[I-1] endif +
    if I=N-1 then W else A[I+1] endif
endall
```

which could be translated as follows:

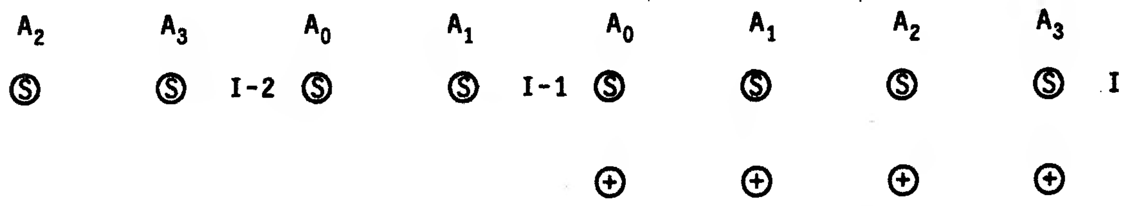
Fig. 7.3



7.2 Combining Subexpressions Over Long Distances

We can combine common subexpressions across different iteration cycles even if the distance between them is quite large -- say, larger than the unfolding. Suppose we need to compute the sum of each element of an array and the element 6 positions to the left. If the unfolding and interlace are 4, we have

Fig. 7.4



Now we can introduce 4 variables obeying the following equations on each cycle:

$Q = A_2[I-2]$
$R = A_3[I-2]$
$S = A_0[I-1]$
$T = A_1[I-1]$

They of course need the correct initial conditions for the first cycle. After that, their "new" values for the next cycle need to be

$Q' = A_2[I-1]$
$R' = A_3[I-1]$
$S' = A_0[I]$
$T' = A_1[I]$

$A_0[I]$ and $A_1[I]$ are already available. For the other two, we introduce two more variables:

$W = A_2[I-1]$
$X = A_3[I-1]$

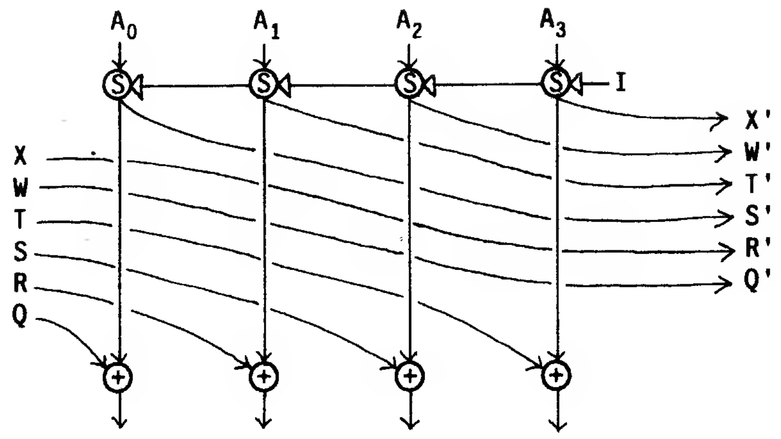
that will provide the right values for Q' and R'. They also need to be initialized for the first cycle, and their "new" values need to be

$W' = A_2[I]$
$X' = A_3[I]$

which are available.

So the body of the loop becomes

Fig. 7.5



Clearly there is a point at which it is not economically feasible to introduce new loop variables, and it is better to recompute the desired data.

7.3 Nested Loops

We can use saved variables to avoid recomputing common subexpressions in any loop, however that loop might be nested. However, in the common case of near neighbor dependencies in all directions in multidimensional arrays, it turns out that only the innermost loop can benefit. Referring back to figure 6.4, we can handle the left and right boundaries only. Introduce variables W, X, Y, and Z with

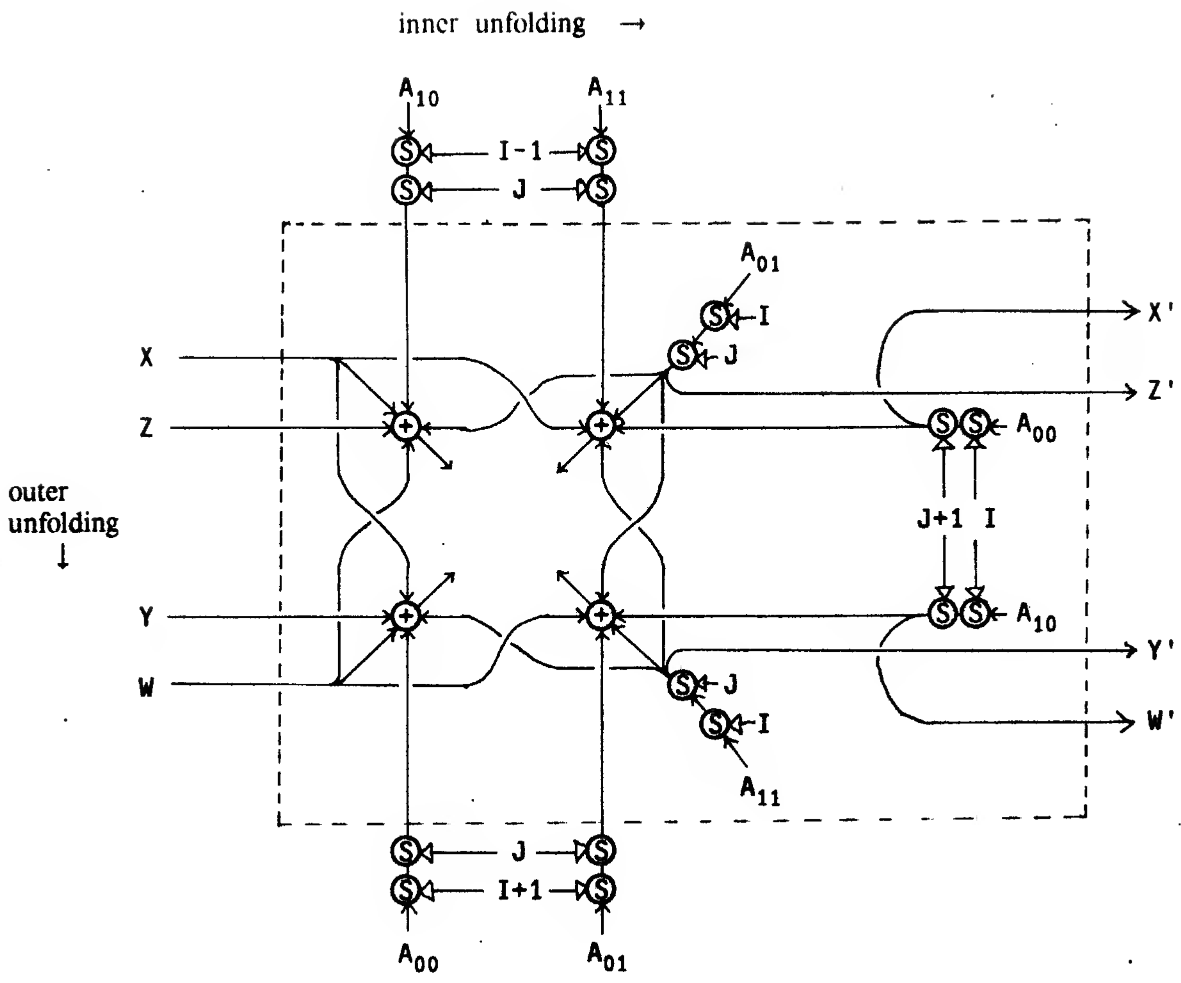
$$W = A_{10}[I,\ J]$$
$$X = A_{00}[I,\ J]$$
$$Y = A_{11}[I,\ J-1]$$
$$Z = A_{01}[I,\ J-1]$$

Then, remembering that J is the inner loop variable, so the next value of J will be J+1 while I will be unchanged, we need

$$W' = A_{10}[I,\ J+1]$$
$$X' = A_{00}[I,\ J+1]$$
$$Y' = A_{11}[I,\ J]$$
$$Z' = A_{01}[I,\ J]$$

So the appropriate graph is

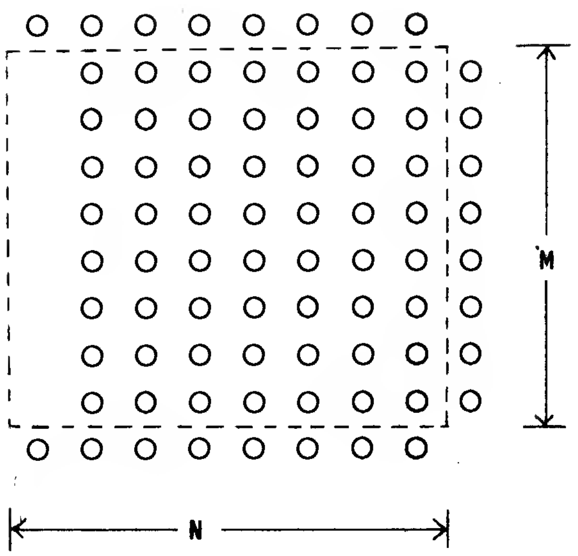
Fig. 7.6



As before, there are 4 array fetches inside the dotted line (though not the same 4). Extra fetches for the boundaries are required only for the dependencies along the outer loop index. The impossibility of combining fetches in this direction is not a serious problem, since the outer loops cycle more slowly -- the time that would be saved by keeping common subexpressions from one cycle to the next, instead of recomputing them, would be insignificant.

The technique of using auxiliary variables to save values changes the analysis of the optimal unfolding shape. Compare figure 6.5 with the new version, using auxiliary variables

Fig. 7.7



The two leftmost columns of fetches are removed. The total number of fetches is now $(M+2)\cdot N$. For fixed product $M\cdot N$, this is least when $N=1$, implying that all unfolding should be in the outer loop, with none in the inner loop. This is somewhat deceptive, however. The total number of array fetches *and* manipulations of saved variables is, as before, $M\cdot N+2\cdot M+2\cdot N$, which is least when $M=N=\sqrt{M\cdot N}$. There is a nonzero cost associated with manipulating these variables, so the best shape is some compromise based on the relative costs of array fetches and manipulation of saved variables.

7.4 Periodic Cyclic Reduction Revisited

We can now exhibit the inner loop of the cyclic reduction algorithm the way it should be written. The version shown in Section 4.9 had an auxiliary variable added by hand.

It should be

```
for K, T := 2*IH+1, Q
do  if K>SIZE+1 then T
    else
        iter K, T :=
            K+2*IH, T[K: Q[K-IH] + A*Q[K] + Q[mod(K, SIZE)+IH]]
        enditer
    endif
endfor
```

It is easier to understand and verify the program in this form.

8. ARRAY CREATION OPERATORS

Up to this point we have used SELECT as the standard array operator for our consideration of interlace and unfolding. Some array creation operators (applicative, of course) are clearly required.

For the cases we are presently interested in (uniform, repetitive operations), the operator that will do what we want is one that adds one more element to an array. This is applied repeatedly to a loop variable that is initially set to an empty array. The *addh* operator of VAL does this conveniently. The size of an array is part of its data in VAL, so such an operator is possible, without the need to tell it the index at which the new element will be stored. However, a language might not require such an operation, or the computer might not provide it. At a slight sacrifice of source program simplicity, we could use an operator that is specifically told at what index to store the element, such as the **append** operator of VAL. Now the semantics of VAL arrays and the **append** operator state that, if if an element is written out of the existing bounds of the array, the bounds are stretched to accommodate the new element. This makes it possible to start with an empty array, whose bounds are set to be right next to where the first element will be written, and write the elements at consecutive addresses.

It would not be wise to assume that we have the exact semantics of the VAL array operators, either in the source language that we wish to support or in the hardware. Several of the details of the VAL operators are not relevant to the present work. For example, the hardware might require that the array bounds be specified at the start of the loop, and perhaps that the array be initially filled with suitable dummy values. It might be useful to "flatten" multidimensional arrays in such a way that the elements will not be written consecutively, even though we know that all positions will eventually be filled.

Therefore, we will assume the minimum possible detail in the array operations, and use them in the most "standard" way possible. For the types of array-creating loops that we are trying to get high performance out of, we will assume that the array bounds are known at the start of the loop,[1] the index of each element is known as it is added to the array, and consecutive additions are most efficient. If these requirements are not met, we assume that the hardware will be less efficient in executing the resultant code. So, for our purposes, the **append** operator will be quite suitable, and we will ignore fine points of its semantics and assume that an actual compiler for a real machine will fill in the details. We will also use "Λ" as the all-purpose empty array that always does the right thing when used as the starting point for a series of **append**'s.

Of course, the exact choice of operations to be made available to the programmer, and the exact instructions that the hardware is capable of executing, will have a strong influence on the efficiency of the system and on code generation techniques when *random* array operations are used. If array A is produced by a regular structure, and a random **append** operation is invoked, as by the VAL assignment

```
    B := A[13: 3.1416] ;
```

then we need to know a lot about the semantics of the operation and the behavior of the hardware before we can begin to design a code generator. But, for regular, repetitive sequences of operations, which we are interested in, we do not need to be concerned with details. In the examples to follow, we will generally use the **append** operator in order to make the index explicit. When **addh** or **addl** appears in the source program, we assume the compiler does the appropriate thing.

1. If the machine is going to support true dynamic arrays, there will be cases in which the final array size is *not* known when the loop starts. However, it might be reasonable to allow the creation of arrays to be less efficient in such cases, so we will provide the bounds and refrain, where possible, from using whatever features the hardware might have for the support of dynamic arrays.

8.1 Structure of Loops that Create Arrays

The canonical way to create a one-dimensional array with a single loop is something like this:

```
%%% set A[I] = f(I)
for A, I := Λ, 0
do  if I=N then A
    else iter A, I := A[I: f(I)], I+1 enditer
    endif
endfor
```

Of course a VAL *forall* could be used, which is roughly equivalent to the above.

Since we use the "vector of vectors" model of arrays, the canonical way to create a two-dimensional array with a double loop is simply to nest the above loop, something like this:

```
%%% set A[I, J] = f(I, J)
for A, I := Λ, 0
do  if I=N then A
    else
        let NEWROW :=
            for B, J := Λ, 0
            do  if J=M then B
                else iter B, J := B[J: f(I, J)], J+1 enditer
                endif
            endfor
        in iter A, I := A[I: NEWROW], I+1 enditer
        endlet
    endif
endfor
```

and analogously for higher dimensions. Nested VAL *forall*'s are roughly equivalent to the above.

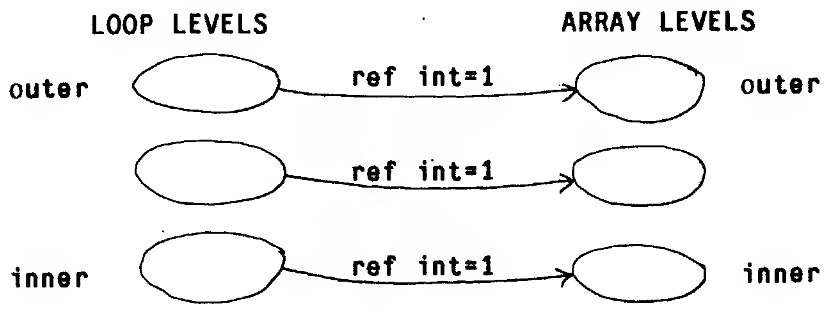
Now this looks ugly and cumbersome. It appears difficult to write source programs this way, and it appears that the underlying machine instructions will be inefficient, since they require packing array descriptors into other arrays.

As for the first objection, one can imagine linguistic constructions that would look better. Where there is no sequential data dependency, the *forall* is a good construction. If nested sequential loops are actually required, some way of assembling the array without explicitly building rows would be useful. Since the VAL language has no such convenient feature, we will use the present notation and leave clean notations to the imagination.[1]

As for the second objection, remember that the vector-of-vectors model is used only when dealing with loop unfolding and array interlace. Once the array slices are chosen, they will be "flattened", so the actual machine instructions that perform array operations will "see" only one-dimensional arrays.

As things now stand, the array dimension structure must match the loop nesting structure. That is, the innermost loop must assemble the one-dimensional vectors, which the next outer loop assembles into 2-dimensional structures, and so on. The loop-array correspondence graph must look like this:

Fig. 8.1



There can be no crossing of the arrows.

1. VAL does allow nested array appends, as in `A[I, J: f(I, J)]`, but it doesn't actually do the correct thing with them as presently defined, and so this feature is not used. It only cleans up a small part of the mess in any case.

What happens if we specifically want to cross the arrows, that is, to create an array that is the transpose of what the canonical nested loop produces? There is really no reasonable notation in VAL to specify such a thing directly. Hence we will ignore the possibility. We will not deal with it because

1. If a notation were created to allow convenient construction of such loops, we know that we will be able to produce efficient machine code. When the slices are flattened, the mapping function that determines where each item is to be written in the slice will simply be different.

2. In the absence of a good notation, we can concoct a bad notation to get the desired effect by creating the array in the normal way and then explicitly transposing it. We would transpose it with something like

   ```
   B := forall I in [0, M], J in [0, N] construct A[J, I] endall ;
   ```

   which we can translate efficiently.

One final observation about the structure of array-creating loops: the reference interval is always 1 (or -1). That is, consecutive items are written, one per loop cycle. If the index at which array elements were being written advanced less often than every cycle, some of the elements would later be overwritten, and the loop could presumably be transformed to remove the superfluous cycles. On the other hand, if the index increased by more than one per cycle, "holes" would be left in the array. We could presumably rescale the index (and rescale everything that reads the array) to squeeze out the holes. Whether a compiler ought to do these things is a question that we will not address.

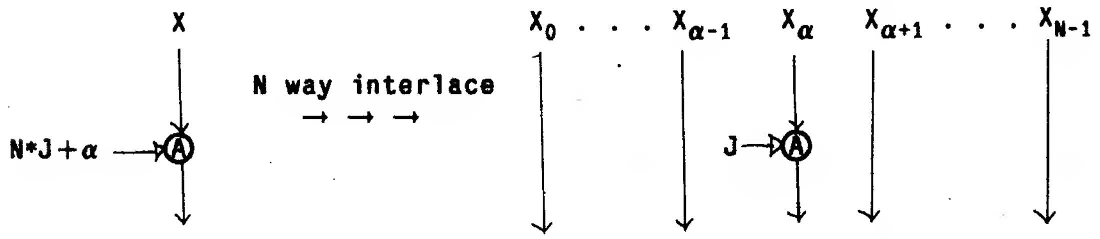
Incidentally, it is the fact that the reference interval is 1 or -1 that makes the VAL *addh* and *addl* operators so useful.

8.2 How to Use the Append Operator

The property of the **append** (and *addh/addl*) operator that makes it trickier than select is that it emits a modified array as an output, and so the order in which an array or array slice passes through a series of **append**'s may be significant. Fortunately, the properties of this operator, when the order in which a series of them is applied is reversed, is fairly simple if we know the array indices. The same array index information that makes it possible for a compiler to perform interlacing and other optimizations allows it to optimize **append** operations.

First, note that a straightforward loop unfolding preserves the order in which **append** operators are applied. (This is because loop unfolding is faithful -- it preserves the application order of *all* operations.) If we have a doubly nested loop, and we wish to perform a 2*2 loop unfolding on it, we just do so, letting the operations go where they have to go. All of the previous analysis for **select** operations, involving unfolding, interlace, and reference interval, apply to **append** operators. When dealing with interlace, we must make the following construction for **append** operators. If the index is known to refer to a particular slice, the operator is put in that slice, and the other slices are sent through untouched.

Fig. 8.2

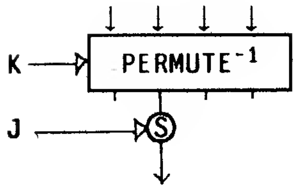


(α is a constant, $0 \leq \alpha < N$)

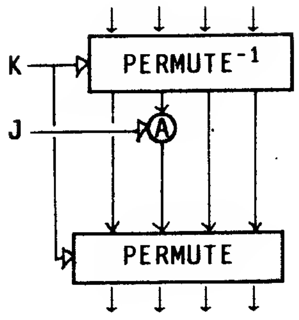
If the index does not behave so well, all slices to which the index could refer must go through a permuter. How to do this was described in Section 4.1 for **select**'s. For **append** we need another permuter to get the array slices back into the correct order from the operator outputs.

Fig. 8.3

if a select requires



an append requires

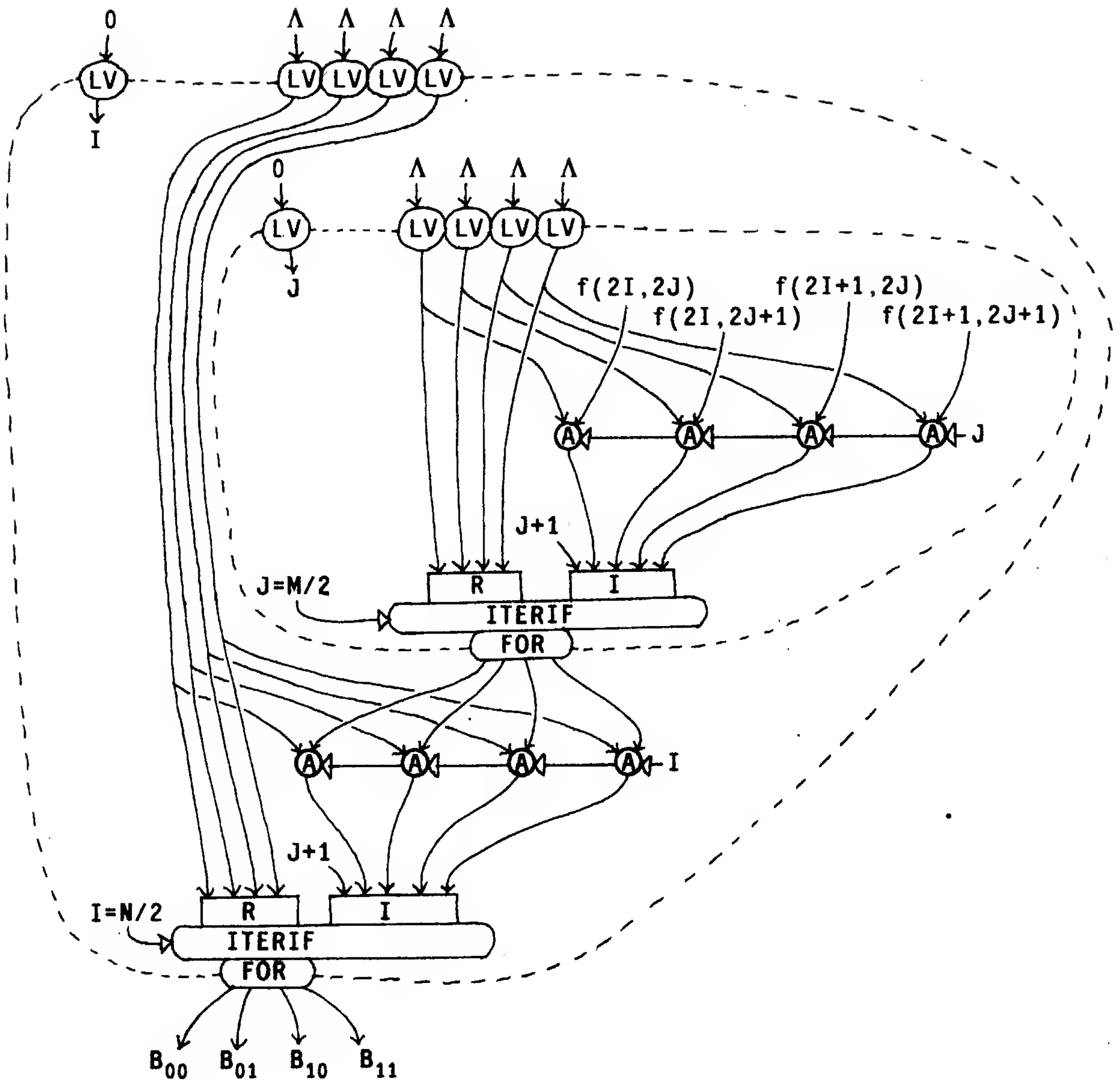


If the loop nesting structure matches the array structure (that is, the arrays are created in the canonical way as described in Section 3.11), re-ordering of operations should not be required. If not, the program may have an apparent data dependency that must be removed. Consider first the canonical double loop:

```
%% create A[I][J] = f(I, J)
%% assume M and N are multiples of 2
for I, A := 0, Λ
do  if I = N then A
    else
        let NEWROW := for J, B := 0, Λ
            do  if J = M then B
                else iter J, B := J+1, B[J: f(I, J)] enditer
                endif
            endfor ;
        in iter I, A := I+1, A[I: NEWROW] enditer
        endlet
    endif
endfor
```

If we perform a 2*2 unfolding, whether we interlace or not, the inner control structures can be coalesced, so the resultant graph appears to have just two loops, one inside the other. If 2*2 interlace is used, the result looks very simple. (Note that the inner loops' control structures can be coalesced, as commonly occurs for nested loops of this type.)

Fig. 8.4



The presence or absence of interlace does not affect the ability to perform 2*2 unfolding efficiently.

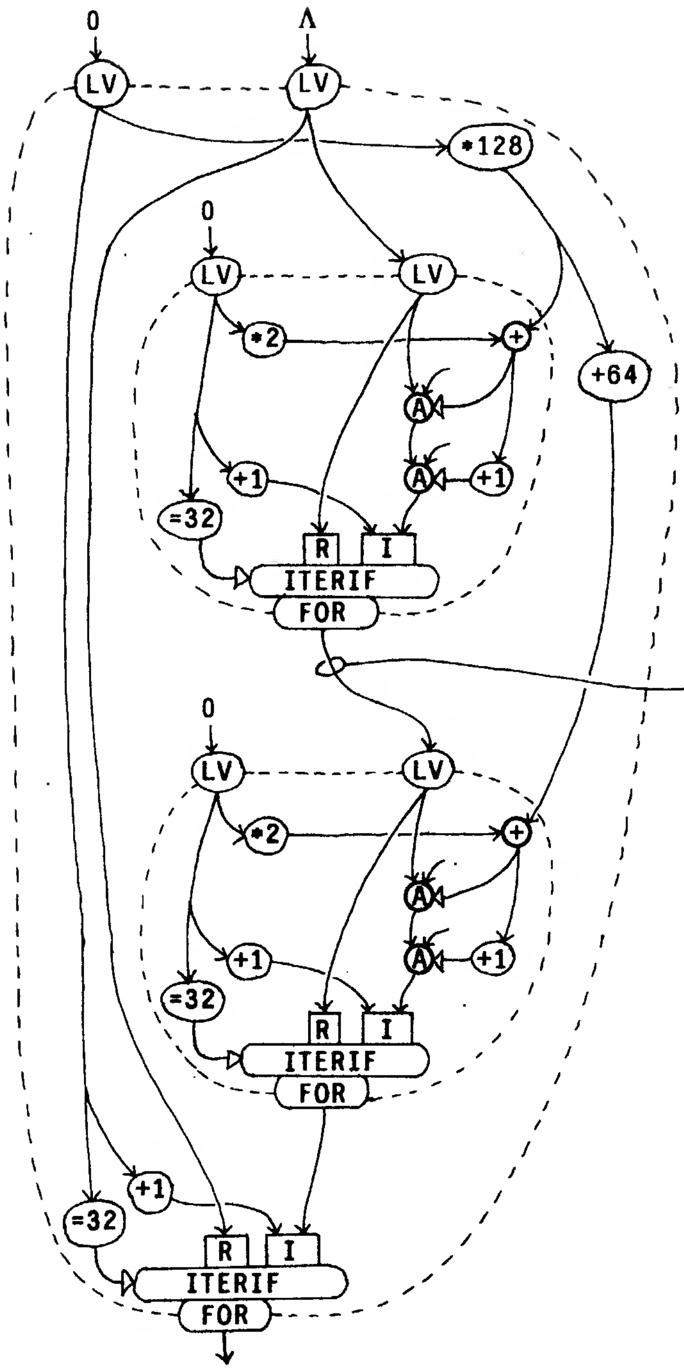
Now if we try to construct an array that does *not* match the loop structure, there is trouble.

```
%% create A[64*I+J] = f(I, J)
%% assume M and N are multiples of 4
for I, A := 0, Λ
do  if I = 64 then A
    else
        let NEWA := for J, B := 0, A
            do  if J = 64 then B
                else iter J, B := J+1, B[64*I+J: f(I, J)] enditer
                endif
            endfor ;
        in iter I, A := I+1, NEWA enditer
        endlet
    endif
endfor
```

The outer loop concatenates all the results of the inner loop into one long array. This is similar to the "flattening" that will be discussed in Chapter 11, but not as benign.

The problem is that the transmission of the single array from one inner loop to the next appears to create a data dependency among the inner loops. If we perform a 2*2 loop unfolding, we can't coalesce the inner loop bodies.

Fig. 8.5



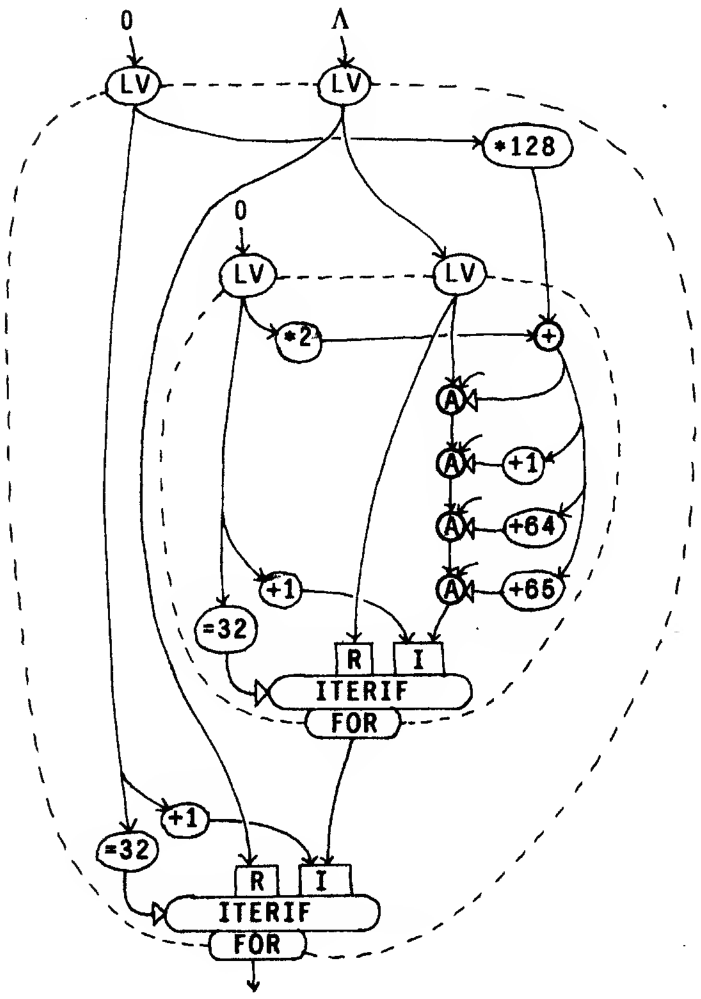
the computation of f
is not shown

this prevents coalescing
the inner loops

No interlace less than 128 will make this dependency go away (and an interlace of 128 would create the need for huge permute operations).

A simple re-ordering will solve things nicely. Since the 4096 **append**'s all take place at different indices, they can be rearranged in any order. The following is a useful order:

Fig. 8.6



Whenever multiple loop unfoldings are being performed and the inner loops are being checked for dependency to see if they can be coalesced, we must look for arrays being **append**'ed to. If an apparent dependency involves an array, we check whether all of the indices involved in the loop unfoldings are disjoint. In the example just given, the question is whether, for each I in [0, 31] and J_1 and J_2 in [0, 31] with $J_1 \neq J_2$, we have

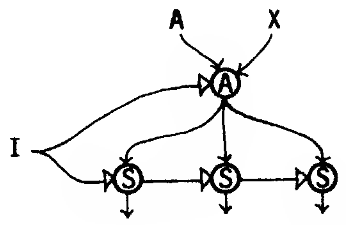
$$(\{128I+2J_1\} \cup \{128I+2J_1+1\} \cup \{128I+2J_1+64\} \cup \{128I+2J_1+65\}) \cap$$
$$(\{128I+2J_2\} \cup \{128I+2J_2+1\} \cup \{128I+2J_2+64\} \cup \{128I+2J_2+65\}) = \emptyset$$

If this is so, the dependency is removed and the loops are coalesced, with the **append**'s all strung together inside the coalesced loop.

8.3 Common Subexpressions Using SELECT and APPEND

The fact that `(A[I: X])[I] = X` means that these two expressions can have "common subexpression combination" performed on them as though they were identical expressions. If a fragment of a graph looks like

Fig. 8.7



we can combine all of the **select**'s with the **append**, and have them just return the data value (`X`) that went into the **append**. This optimization is treated just like the common subexpression combination of Chapters 6 and 7. As in those chapters, this will be most useful in iterations when the common subexpressions come from different cycles of the unfolded loop. If those different cycles occur in the same unfolding of an unfolded loop, the combination is straightforward. If in different unfoldings, then, as in Chapter 7, we compute the common value in the earliest unfolded cycle and feed it forward to the later cycles. In a situation in which an **append** is one of the things being combined, the **append** will always be in the earliest cycle, so that it will be the source of the data, and the **select**'s will all be consumers. This follows from causality -- one can't read a value out of an array before it has been written.

Combining **append** and **select** usually occurs in algorithms that solve a "recurrence relation" -- computing elements of an array sequentially using, for each element to be computed, the value stored in the previous one or more elements of the same array. For example, if we want to set

```
B[I] = B[I-1] + A[I]     for 1 ≤ I ≤ 64
```

This must be computed by a sequential loop[1]

```
for I, B := 1, [0: B0]          % need initial element
do  if I=65 then B
    else iter I, B := I+1, B[I: B[I-1] + A[I]] enditer
    endif
endfor
```

Digression

The optimization we are about to exhibit has been discussed in the literature [31] in the ongoing debate over the suitability of applicative languages. The optimization basically consists of introducing a new loop variable whose value is `B[I-1]` on each cycle. If the source program were so transformed, we might have

```
for I, B, LASTB := 1, [0: B0], B0
do  if I=65 then B
    else
        let NEWB := LASTB + A[I]
        in iter I, B, LASTB := I+1, B[I: NEWB], NEWB enditer
        endlet
    endif
endfor
```

This sort of transformation can of course be made whether the program is written in FORTRAN, VAL, or nearly any language. Experienced programmers frequently do make such a transformation if they know that it will improve performance on the target computer that they will be using. On many conventional machines there will be little, if any, improvement, because array references aren't particularly expensive (that is, if the compiler keeps the index variable in an index register). On some array and pipeline machines it is an important transformation to make, because the last item written into B may be somewhere inside the pipeline, rather than in the memory, when it is needed, and it cannot be obtained without destroying the pipelining.

1. This *particular* computation could be rewritten so that it does not require a sequential loop, but we are interested in general recurrences of this kind, for which no shortcut is possible.

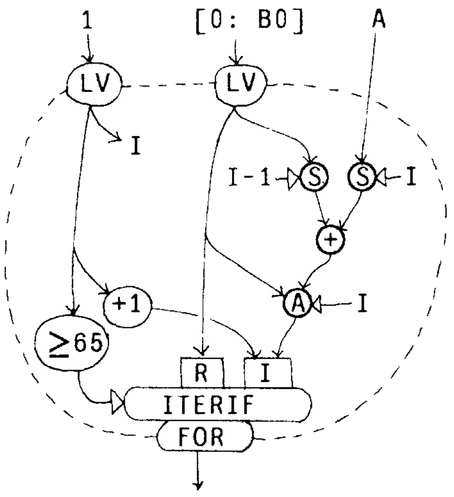
On a data flow machine it is also an important transformation, because array references are expensive.

In any case, the programmer should not have to make the transformation by hand -- the compiler should do it. By doing things this way, the compiler for each target machine can optimize its code for that machine, and fewer machine-dependent quirks will be seen in source programs.

End of digression

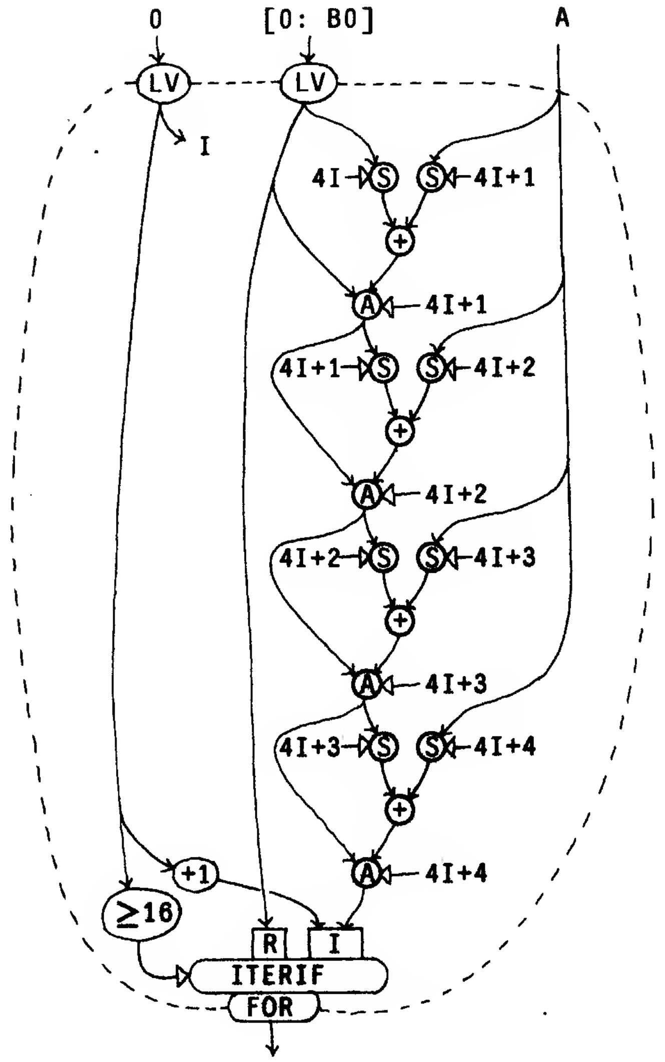
The original graph is

Fig. 8.8



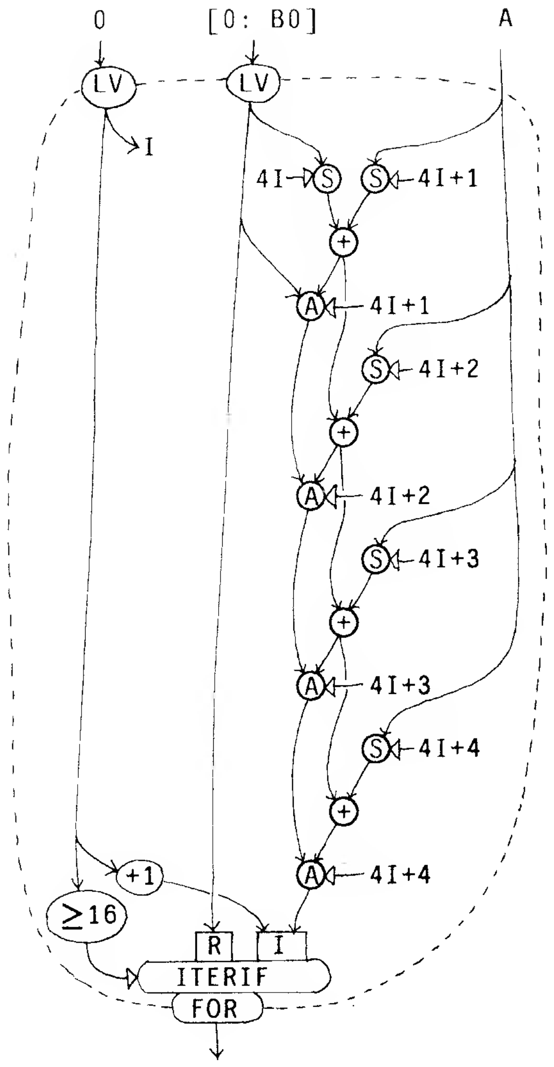
We first perform a 4-way unfolding and scale the loop variable

Fig. 8.9



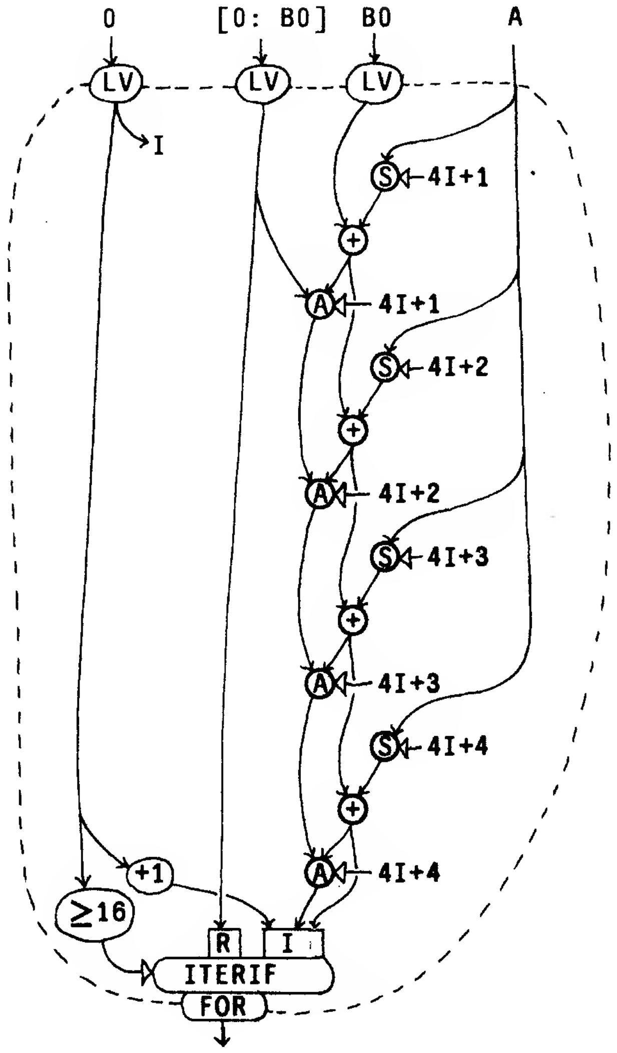
Now common subexpressions are removed in the same unfolded cycle, using the knowledge that we have about **append** operators.

Fig. 8.10


0
[0: B0]
A
LV
LV
I
4I
S
S
4I+1
+
A
4I+1
S
4I+2
+
A
4I+2
S
4I+3
+
A
4I+3
S
4I+4
+
A
4I+4
+1
≥16
R
I
ITERIF
FOR

Now a common subexpressions -- B[4I] -- is optimized across cycles, using an auxiliary variable.

Fig. 8.11



As a final step we might interlace the arrays. We could have interlaced before performing the common subexpression removal and introducing the loop variable -- in previous examples we did so. The order in which the transformations are made does not matter.

9. ARRAY REFERENCES WITH UNKNOWN REFERENCE INTERVAL

This chapter considers the case in which the reference interval is not known prior to execution because it depends, from one loop cycle to the next, on the ongoing calculation. The techniques to be shown work well only if the reference interval, while unknown, is known to have a small upper bound that can be determined at compilation time. If the upper bound is large, many array references will be wasted and the code to manipulate arrays will overwhelm the rest of the computation.

Loops for which the reference interval has a small upper bound are important in practice. In fact, the most important case is that in which the upper bound is one, that is, the array index increases either by zero or by one on any given loop cycle. This case includes the programs in which some "pointer" is scanning across an array and, on any given cycle, either moves on to the next element or stays unchanged.

The most common example of this is the "merge" part of the "mergesort" algorithm:

```
%% A, B = incoming sorted arrays
%% ASIZE, BSIZE = their sizes
%% AI, BI = pointers for scanning them
%% OUT = array being constructed
%% OI = pointer for constructing it
for AI, BI, OI, OUT = 1, 1, 1, Λ
do  if AI > ASIZE & BI > BSIZE then OUT
    elseif BI > BSIZE | (AI ≤ ASIZE & A[AI] < B[BI]) then
        iter AI, OI, OUT := AI+1, OI+1, OUT[OI: A[AI]] enditer
    else iter BI, OI, OUT := BI+1, OI+1, OUT[OI: B[BI]] enditer
    endif
endfor
```

Another rather interesting such problem is the "Exercise attributed to R. W. Hamming" in [27]

```
for A, I, P2, P3, P5 := [1: 1], 2, 1, 1, 1
do  if I > N then A
    else
        let X2 := 2*A[P2] ;
            X3 := 3*A[P3] ;
            X5 := 5*A[P5] ;
            NEW := min(X2, X3, X5)
        in  iter A, I, P2, P3, P5 :=
                A[I: NEW],
                I+1,
                if X2 <= NEW then P2+1 else P2 endif,
                if X3 <= NEW then P3+1 else P3 endif,
                if X5 <= NEW then P5+1 else P5 endif
            enditer
        endlet
    endif
endfor
```

We will assume that the uncertain reference interval is expressed as a conditional that chooses one of several constants as the amount by which the index variable is to be incremented, such as

```
iter
    J := if P0 then J+1
    elseif P1 then J+2
    elseif P2 then J+3
    else J                          %increment = 0
    endif
enditer
```

There are, of course, other ways to formulate it, but they differ from this only in some transformations a compiler would make to determine the upper bound on the amount of the increment. By making the amount of the increment a choice of manifest constants, the problem of determining the bound is obviated.

The general case we will consider is that in which the possible values of the increment are unknown but bounded multiples of a known constant. The latter constant will, of course, be a power of two for convenience. It is the *reference multiplier*, and is somewhat analogous to the reference interval of previous chapters. The upper bound on the unknown number by which it is multiplied is the *reference bound.*

For example, if the increment is known to be 0, 4, or 12, the reference multiplier is 4 and the reference bound is 3. One could, of course, declare the reference multiplier to be 1 and the bound to be 12, but it would result in very inefficient optimized graphs.

The general technique for handling an unknown reference interval in the presence of loop unfolding is as follows: The multiple loop instantiations need values from the array for several loop cycles at a time. When the reference interval was known, we were able to predict exactly which elements would be needed and to obtain those elements in advance. In the present situation, we can only predict that the needed elements will come from a certain set, and obtain *all* of the elements in that set. Then, using the actual values of the reference interval that become available as the loop cycles progress, we can route the correct elements to the place where they are needed and throw the others away.

Because of the unknown "phase" of the array indices with respect to the loop instantiations, we are much more likely to require permuters than in the case of a known reference interval: Permuters are needed whenever the interlace is greater the the reference multiplier. (They were formerly needed only when the interlace was greater than the product of the reference interval and the unfolding.)

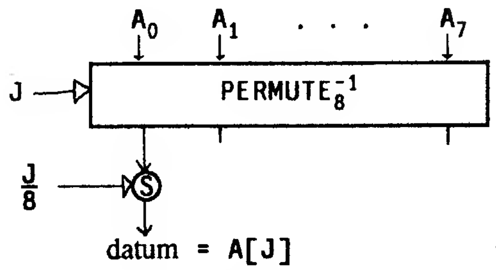
We begin by using the techniques of Chapter 4 as though the loop unfolding were only one and the "reference interval" of Chapter 4 were equal to the present reference multiplier. That is, we generate the data flow graph for accessing the element needed by the first instantiation only. This will require a permuter if the interlace is greater than the reference multiplier.

Example:

```
interlace = 8
reference multiplier = 1
reference bound = 2
unfolding = 4, but the initial graph will be made as though it were 1
```

To get `A[J]` for the first instantiation, we have

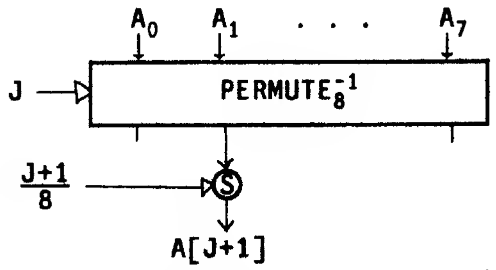
Fig. 9.1



Now, since the unfolding is 4, we need the data for the next 3 cycles. Normally, the datum for the next cycle would be produced by the same graph as above, but with index J increased by 1.

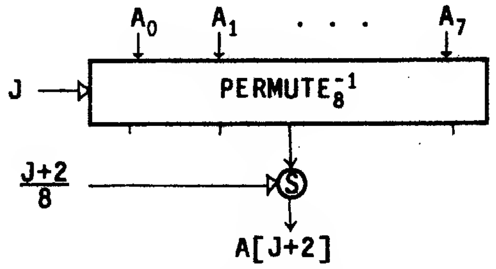
We might do it this way:

Fig. 9.2



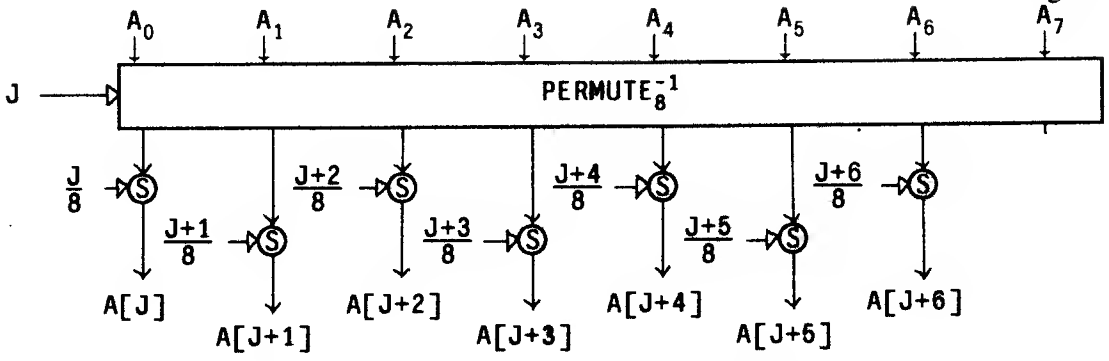
and the graph for the datum after that would be

Fig. 9.3



Because the reference bound is 2, the datum that we want for the second instantiation could be any of `A[J]`, `A[J+1]`, or `A[J+2]`. Since the unfolding is 4, 3 instantiations are needed after the initial one, which could require any of `A[J]` through `A[J+6]`. To avoid long chains of data dependencies, they can all be computed in parallel, using just one **permute** operator.

Fig. 9.4



This construction, to simultaneously access many consecutive (relative to the reference multiplier) items will be called a *multi-select.*

Incidentally, it is reasonable to suppose that the operation of computing things like (J+5)/8 and using the result as a select index will be supported by very efficient hardware mechanisms.

Since we may need A[J] through A[J+6], this construction matches our needs well. The reason is that, for this case, we have

```
reference multiplier * reference bound * unfolding = interlace
```

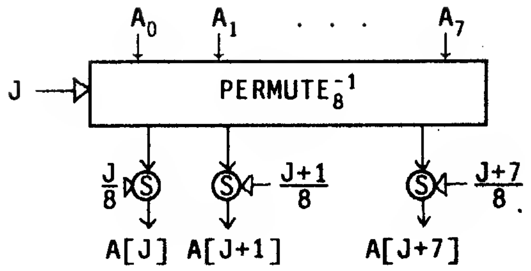
This is the optimum situation. If the product on the left were greater than the interlace, each slice would have to go to multiple array operations, just as in Chapter 4. Unlike Chapter 4, however, we still need a permuter. We always need a permuter when the interlace is greater than the reference multiplier. That permuter must permute $\frac{\texttt{interlace}}{\texttt{reference multiplier}}$ items.

For example, if

```
interlace = 8
reference multiplier = 1
reference bound = 4
unfolding = 4
```

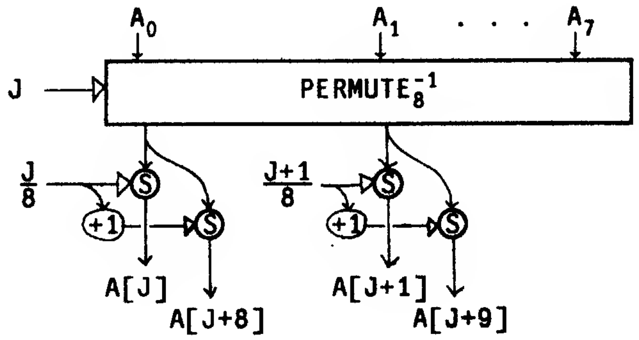
then this

Fig. 9.5



will get `A[J]` through `A[J+7]`. But we need data up through `A[J+12]`, so most slices must go to two select operators.

Fig. 9.6



9.1 Reference Multiplier Not Equal to One

If the reference multiplier is not equal to one, it affects things the same way a reference interval not equal to one affected things in Chapter 4: The "effective interlace" becomes $\left\lceil \frac{\texttt{interlace}}{\texttt{reference multiplier}} \right\rceil$. This is the number of array slices required by the computation.

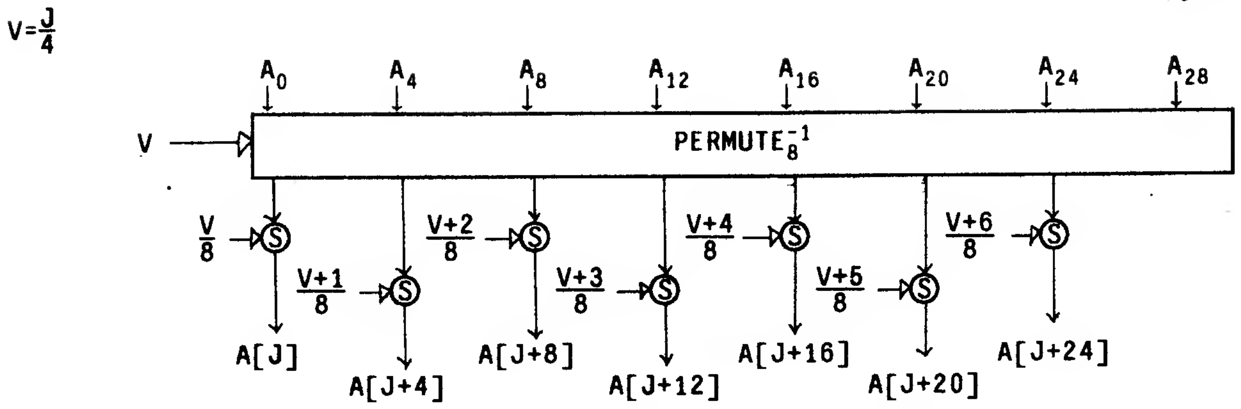
Example:

```
interlace = 32
reference multiplier = 4
reference bound = 2
unfolding = 4
```

We can rescale the loop index, dividing it by the reference multiplier. Let V be the scaled index, so `V = J/4`. Then the situation, in terms of V and array slices A_0, A_4, ... A_{28}, looks just like the earlier case with interlace = 8 and reference multiplier = 1.

The "multi-select" looks like:

Fig. 9.7



9.2 Using Multi-selects to Provide the Data

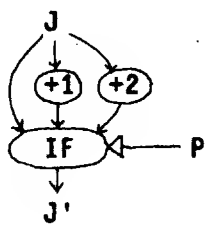
Recall that the program fragment controlling the index variable look generally like this:

```
iter J :=
    if P1 then J
    elseif P2 then J+1
    else J+2
    endif
enditer
```

There is some conditional (or huge tree of conditionals) computing the new value of the index by selecting fixed increments to add to it. Those increments lie between zero and the reference bound, assuming that the index has been rescaled if the reference multiplier is not 1.

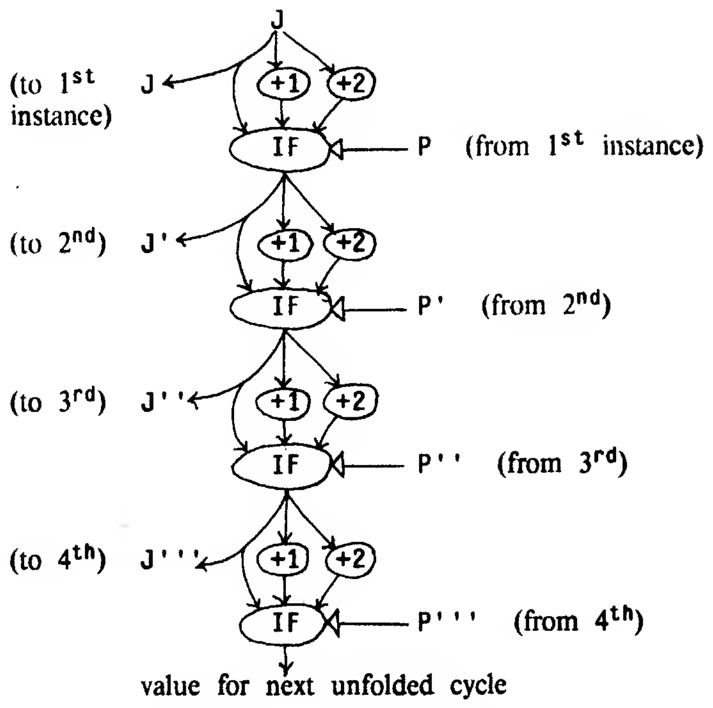
Graphically, this looks something like this:

Fig. 9.8



These are cascaded as many times as the unfolding requires:

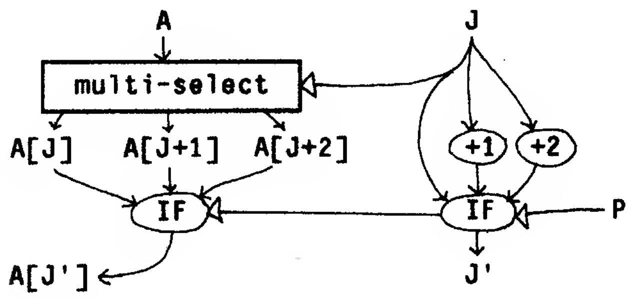
Fig. 9.9



Now the first instance requires A[J]. That comes out of the multi-select easily. The next instance requires A[J'], which is A[J], A[J+1], or A[J+2], depending on a computation that uses P. To obtain A[J'], we duplicate the same conditional, controlled by P, but with A[J], A[J+1], and A[J+2] entering it instead of J, J+1, and J+2.

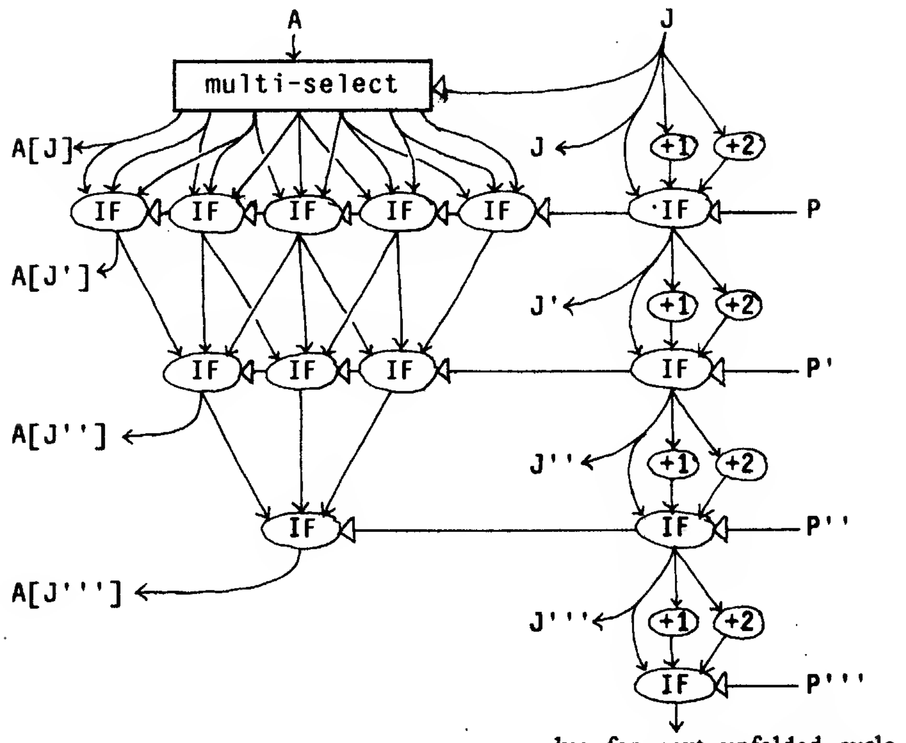
The result is:

Fig. 9.10



The graph for the entire loop with unfolding of 4 is:

Fig. 9.11



value for next unfolded cycle

Unfortunately, this consumes an amount of space that is quadratic in the amount of unfolding.

9.3 Array Appends with Unknown Reference Interval

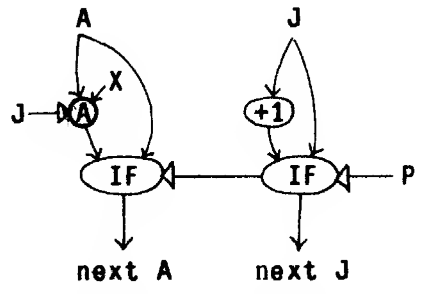
When a loop needs to "write" to an array, that is, send the array through **append** operations, and the reference interval is unknown, a similar sort of construction is used. For each unfolded loop body there is a range of array indices into which **append**'s might take place. Within that range it is not known in advance which indices will be written to and which will be left unchanged, so a "multi-select" is used to read the original contents of all elements in that range. Those values are then sent through a network of conditionals to substitute the new values where needed, and the results go to a "multi-append" to write them back into the array. The design of the "multi-append" is similar to that of the "multi-select". The network of conditionals that inserts the new values is somewhat similar to the network that performs **select**'s with unknown reference interval. It, too, has a complexity that is quadratic in the amount of unfolding. It will not be shown here.

It happens that a general **append** with unknown reference interval is probably not a useful thing in practice. This is because such a loop, if the reference bound is two or more, must occasionally leave "holes" in the array. A much more common situation is the special case of reference multiplier and reference bound both equal to one, with the **append** taking place only on those cycles in which the loop index actually increases. That is, in any cycle one of two things happens: Either an element is added at the index given by the index variable and the variable is increased by one, or the array and index variables are both left unchanged. The typical iteration control thus looks something like this:

```
iter A, J, <other variables> :=
    if P then A[J: X] else A endif,
    if P then J+1 else J endif,
    <other values>
enditer
```

The graph with no unfolding or interlace looks like:

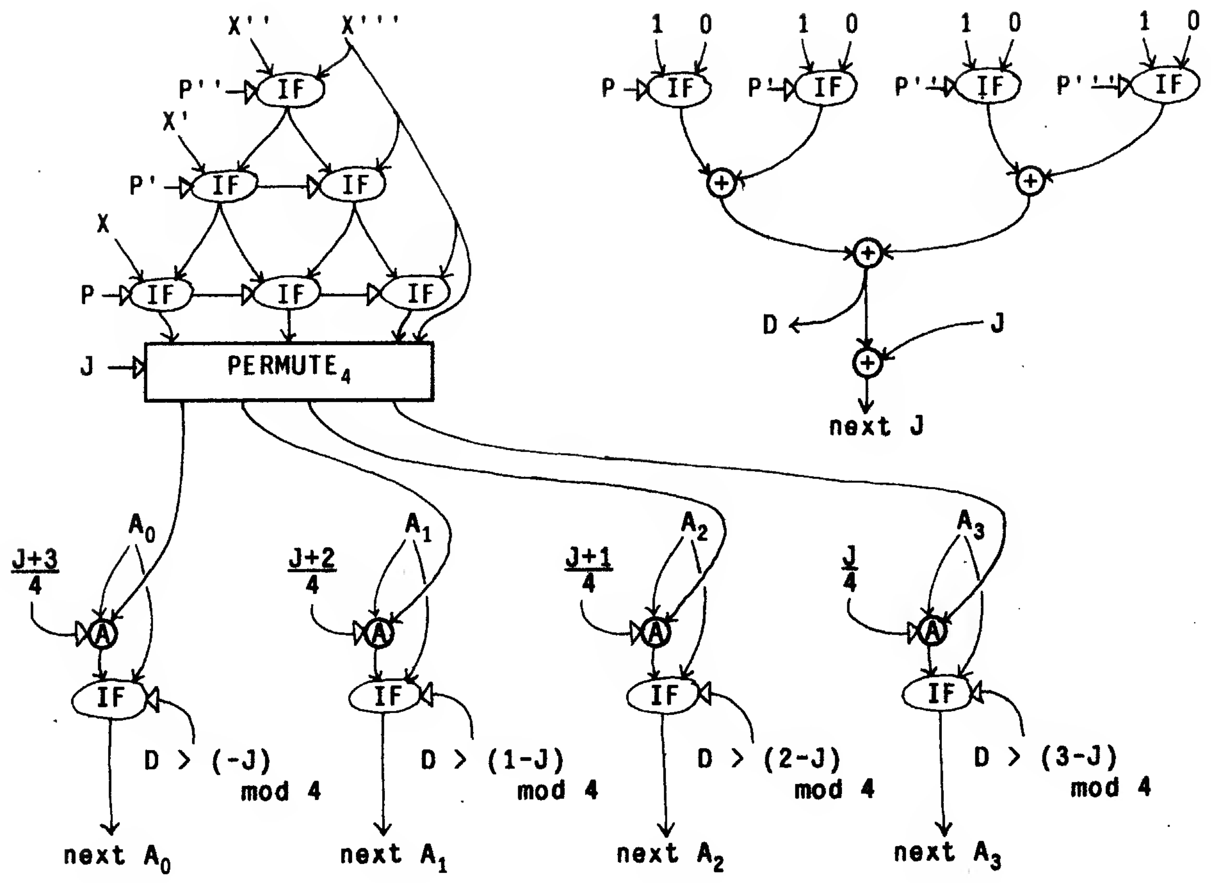
Fig. 9.12



Assuming a unfolding and interlace of 4, we have, in the unfolded loop body, four boolean values P through P''' and four data values X through X'''. (For each P value that is false the corresponding X value will not be used.)

The graph for this is:

Fig. 9.13



10. DISTRIBUTION AND DECOUPLING OF CONTROL STRUCTURES

It was suggested in Section 3.10 that, when inner loop control structures arising from unfolding on an outer loop are coalesced, they should not force the loops to operate in lock-step, but should, in some cases at least, allow them to "decouple". This chapter will show why such decoupling is important, and will suggest how it might be achieved.

10.1 Multidimensional Data Dependencies and the "Wavefront" Transformation

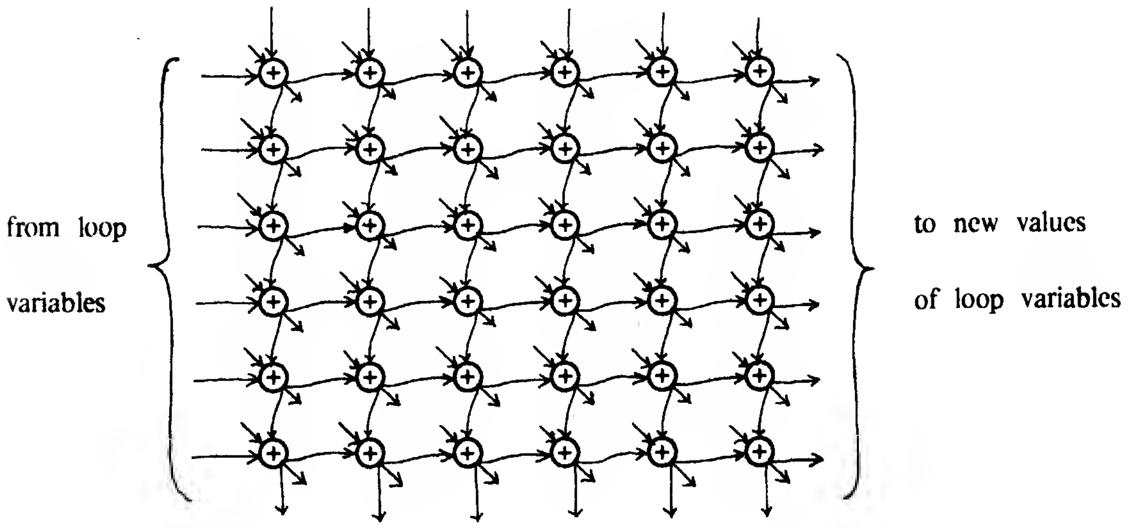
Consider an algorithm which is a double nested loop creating a two-dimensional array. Each element of the result depends, in part, on the values of the *result* array in the adjacent positions above and to the left. This implies some sort of upper-left-to-lower-right computation order, which must be expressed in the source program. The usual way to express it is by explicitly specifying a top-to-bottom and left-to-right "raster scan" order. It is reasonable to express it the same way in a data flow program, as follows:

```
for I, Q := 1, [0: TOP_BOUNDARY_ARRAY]
do  if I=N then Q
    else
        let ROW := for J, R := 1, [0: LEFT_BOUNDARY_SCALAR]
            do  if J=M then R
                else
                    iter J, R :=
                        J+1, R[J: A[I][J]+Q[I-1][J]+R[J-1]]
                    enditer
                endif
            endfor
        in iter I, Q := I+1, Q[I: ROW] enditer
        endlet
    endif
endfor
```

Of course, a data flow or other parallel computer need not enforce a strict raster scan order, as long as the data dependencies are satisfied.

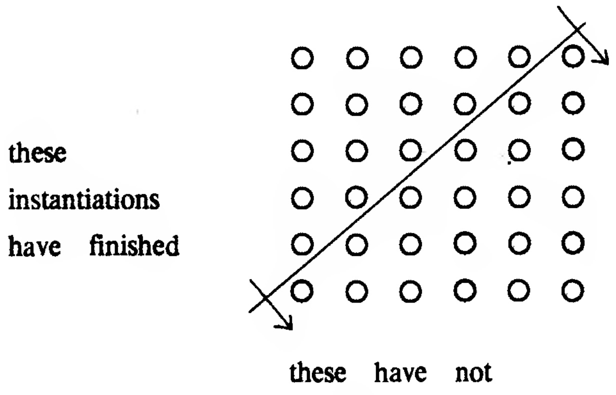
As was indicated in Chapters 6 and 7, the unfolded and coalesced loop instances make a 2-dimensional matrix. When common subexpressions are combined, data flows through that matrix from upper left to lower right.

Fig. 10.1



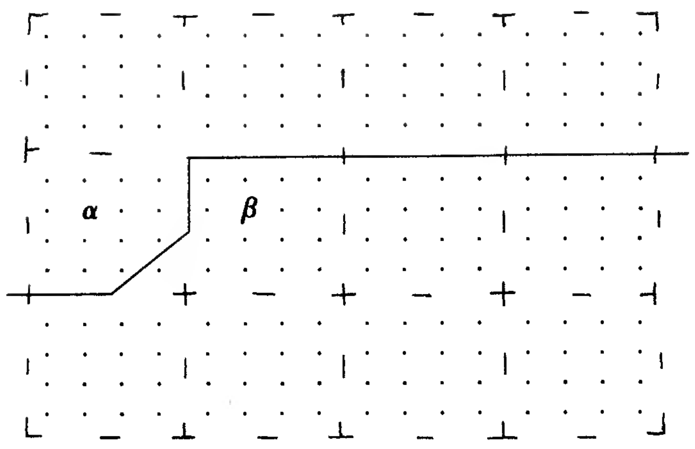
Because of the data dependencies, one would expect the activity in the graph to proceed, approximately, along a "wavefront" -- a diagonal line that sweeps from upper left to lower right. An instantaneous picture of the graph might look like this.

Fig. 10.2



If the unfolded loops all operate in lock-step so that none may begin a cycle until all have completed the previous cycle, the next group of instances may not begin until the current wavefront reaches the lower right corner. Looking at many blocks of the array (refer to Section 3.12) the boundary between processed and unprocessed parts must look like this

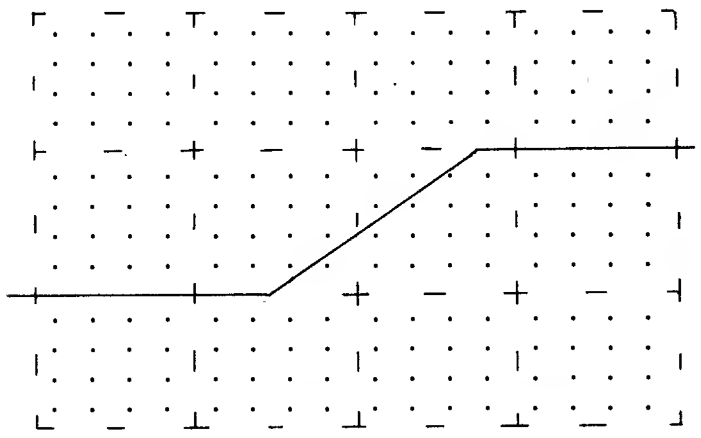
Fig. 10.3



Block β cannot begin until block α has completed.

If, on the other hand, the loops are decoupled, loop instances near the top of a block will be able to start while instances near the bottom of the previous block are still busy. This is because loop variables near the top are recycled before those near the bottom, and a loop cycle can begin as soon as its loop variables become available. The boundary between processed and unprocessed parts now looks like this:

Fig. 10.4



The diagonal part of the boundary slides smoothly to the right, as one would expect in a "natural" computation. The entire array is processed in horizontal strips, the topmost strip being processed first.

Depending on the distance over which data dependencies occur among the loop cycles, and on the amount of unfolding at each level, the overlap among the groups could be much higher than the example here shows. The execution of the decoupled loops could therefore be separated by several entire cycles.

On most vector, pipeline, or parallel processors, the "raster scan" order is not efficient. This is because those machines cannot handle sequential data dependencies among loop cycles (sometimes referred to in the literature as "recurrence relations") well. Optimizers therefore sometimes re-order the computation so that it will proceed along a different raster scan, in this case one rotated 45 degrees from the originally specified order. In this example, a program that originally was, in FORTRAN

```
DO 99 I = 1, N
DO 99 J = 1, M
```

becomes, in effect:

```
DO 99 AA = 1, M+N-1
DO 99 I = MAX(1, AA+1-M), MIN(AA, N)
J = AA-I+1
```

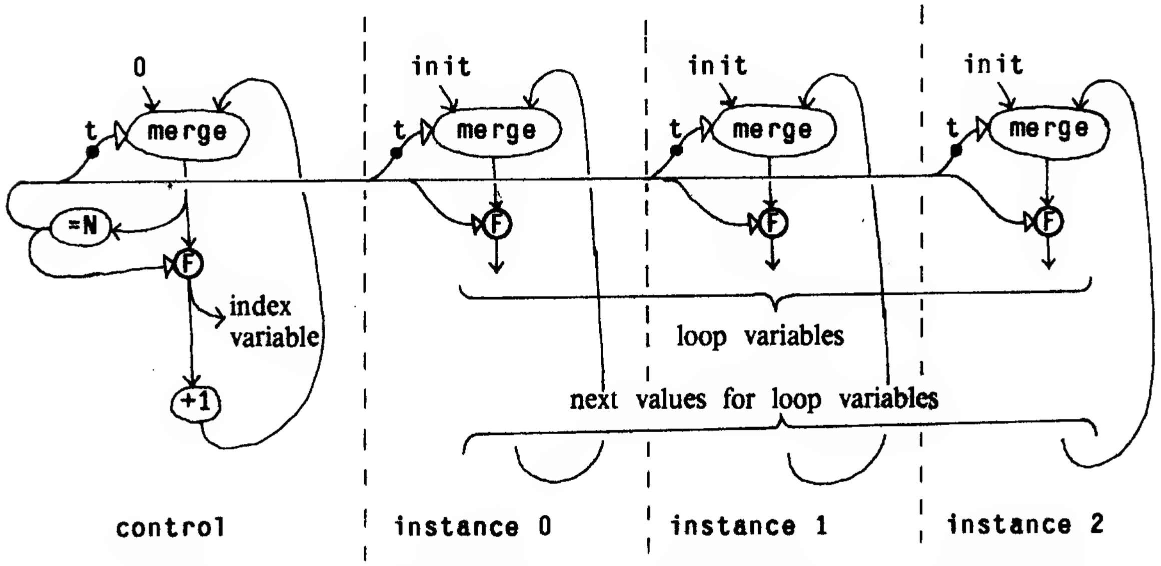
The "coordinates" I and J of the computation space are reparameterized in terms of new coordinates I and AA. The inner loop proceeds along a diagonal line with no sequential data dependencies, which a vector or pipeline machine can handle well. The outer loop moves this line (the "wavefront") downward and to the right.

This "wavefront transformation" plays an important role in optimizers, such as PARAFRASE [38], for conventional supercomputers. It is *not needed* in a data flow system. By allowing loop bodies to decouple, the wavefront will form naturally.

10.2 Achieving the Decoupling

To decouple the multiple inner loops, we must allow the operators that control the recycling of loop variables to function at different times. These operators are controlled by boolean control values that come from the test operators that sense when the loop is to terminate. When the loops were coalesced, all of the loop variable recycling operators were put under the control of a single test operator. An example of such an arrangement is the following:

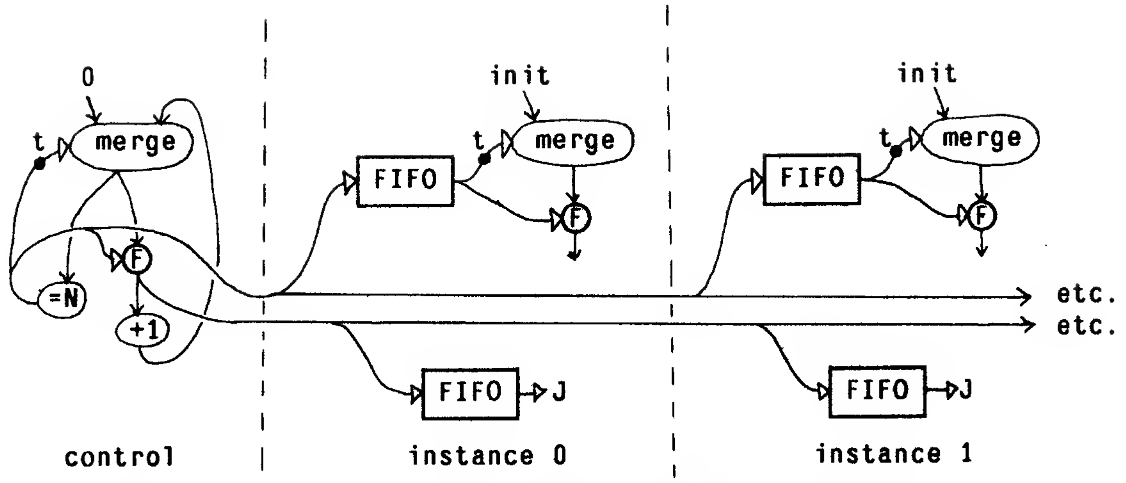
Fig. 10.5



The MERGE and F gates controlling the loop variables are locked into step with each other by the static firing rules: The =N operator cannot emit a boolean token until all of the operators to which it send tokens have absorbed the previous token. On the Dennis-Misunas data flow design [26] this is accomplished by having operators send acknowledgment tokens to indicate that they have absorbed their inputs. Because of this, the MERGE and the various F gates cannot fire at significantly different times.

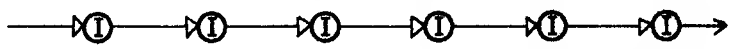
To allow these gates to fire at different times, and hence allow the loop variables to recycle at different times, we can place short FIFO ("first-in first-out") buffers in the lines carrying the control information.

Fig. 10.6



The control logic at the left can now run several cycles ahead of the instances, filling the FIFO buffers. The instances can consume tokens from the buffers at different times, with some instances perhaps running several cycles (limited by the FIFO capacity) ahead of others.

FIFO buffers can be implemented as chains of identity operators:

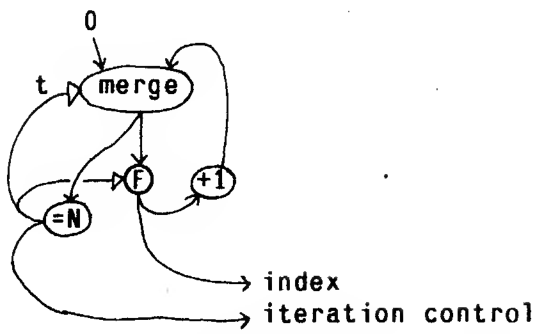


For any but the shortest chains this is a very wasteful way of doing it. There are many ways to implement these buffers in a space-efficient manner. See [25] for a design for efficient huge FIFO buffers, in the context of a machine that uses such buffers for direct storage of arrays.

There is another way to solve the problem of lock-step operation -- replicate and distribute the control structures.

If a separate copy of the control structure

Fig. 10.7



is used for each instance, the instances will be able to run with the appropriate timing differences. Distributing the control structures has another potentially important advantage. It reduces the amount of communication among geographically separated parts of the program. To achieve high speed and avoid bottlenecks, it is important for the instances to be executed in different parts of the computer. The transmission of the index and iteration control tokens (on each cycle) among the parts can create a heavy load on the communication system, and may lead to bottlenecks. Distributing the control structure can prevent this.

Loop control subgraphs such as the one above will probably be very commonplace in compiled data flow programs, and hence it will be useful if such graphs can be implemented efficiently in terms of speed, processor loading, and instruction space. An instruction set design by Burkowski [13] features some instruction types designed for efficient implementation of various loop control schemes such as this one.

11. FLATTENING OF ARRAYS

It has been assumed all along that multidimensional arrays are interlaced independently at each level. It was further assumed that the slices were themselves to be treated as multidimensional arrays. For example, if an array X is 4-dimensional and has A×B×C×D interlace, then for each α in [0, A-1], β in [0, B-1], γ in [0, C-1], and δ in [0, D-1] there is a slice $X_{\alpha\beta\gamma\delta}$ which is itself a 4-dimensional array.

The bounds of each slice can be calculated from the bounds of the original array. If the array X described above has bounds

$$[P_A, Q_A] \times [P_B, Q_B] \times [P_C, Q_C] \times [P_D, Q_D]$$

then a simple calculation shows that slice $X_{\alpha\beta\gamma\delta}$ has bounds

$$[\lceil \tfrac{P_A-\alpha}{A} \rceil, \lfloor \tfrac{Q_A-\alpha}{A} \rfloor] \times [\lceil \tfrac{P_B-\beta}{B} \rceil, \lfloor \tfrac{Q_B-\beta}{B} \rfloor]$$
$$\times [\lceil \tfrac{P_C-\gamma}{C} \rceil, \lfloor \tfrac{Q_C-\gamma}{C} \rfloor] \times [\lceil \tfrac{P_D-\delta}{D} \rceil, \lfloor \tfrac{Q_D-\delta}{D} \rfloor]$$

If the bounds for the given array X are known at compilation time, the bounds for each slice can be computed at compilation time. If the bounds for X are defined in terms of quantities that are known only at execution time, the bounds for each slice can be computed in terms of the same quantities. Furthermore, if the bounds are known at compilation time modulo the interlace, the computation is much less formidable than the formula above would indicate. For example, if P_A is known to be A*J+3, where J will not be known until execution, then $\lceil \frac{P_A-\alpha}{A} \rceil$ = J if $\alpha \geq 3$, or J+1 if $\alpha < 3$. (The value of α, the slice number, is *always* known at compilation time for each slice.)

11.1 Order of Indices in Flattening

Knowing the bounds, either numerically at compilation time or in terms of computed values, we can flatten the array. This is done by choosing an index and taking the one-dimensional strips of the array that are formed when that index varies but all other indices are held constant. There is one such strip for each value of the other indices. Choose a second index and lay end-to-end those strips that arise from letting that second index vary. There is now a collection of longer strips, one for each value of the remaining indices. Each strip has the data from an entire plane of the array. Choose another index and lay the strips end-to-end again, and so on. The result is one strip containing the entire array.

We can choose any order of the indices in doing this. For example, if array slice S has bounds

$$[U_A, V_A] \times [U_B, V_B] \times [U_C, V_C] \times [U_D, V_D]$$

it could be flattened into the one-dimensional slice T with

$$T[(V_D-U_D+1)((V_C-U_C+1)((V_B-U_B+1)(I)+J)+K)+L] = S[I, J, K, L]$$

This order is sometimes called "last index varying most rapidly" and is the order we will generally use. One could flatten in the opposite order, obtaining R with

$$R[(V_A-U_A+1)((V_B-U_B+1)((V_C-U_C+1)(L)+K)+J)+I] = S[I, J, K, L]$$

sometimes called "first index varying most rapidly" (the way arrays are required to be organized in FORTRAN), or in any other order.

We could leave holes in the flattened array by replacing the multipliers (the expressions like "V_C-U_C+1") with any higher number. This could be useful if the bounds of different slices were different, and we wanted to use the same multipliers for all slices.

Knowledge of the bounds on the original slices, whether they be computable at compilation time or defined in terms of execution time values, can be turned into knowledge of the bounds of the flattened slices.

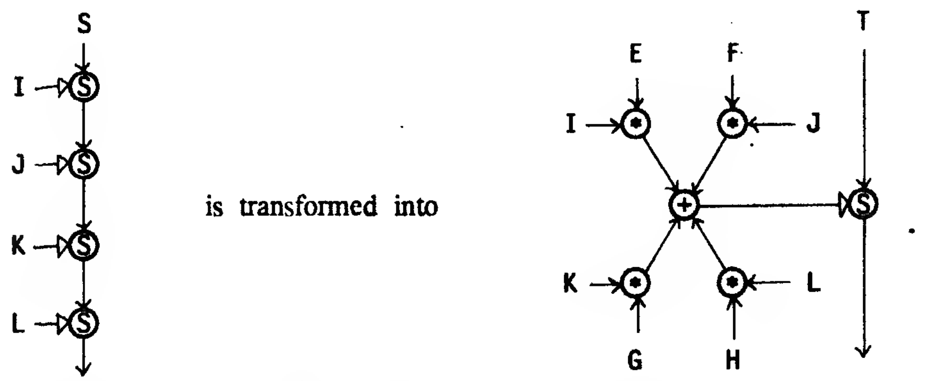
11.2 Using Flattened Slices in SELECTs

Each K-dimensional array slice will presumably be used in **select** operations, in which the slice will go through K individual select operators to yield a scalar. When the slice is flattened, it becomes a 1-dimensional array, which therefore needs to go through just one **select**. The index value for that **select** can be found from the correspondence between the original slice and the flattened one. This correspondence is a simple linear combination, such as

```
S[I, J, K, L] = T[E·I + F·J + G·K + H·L]
```

where S is the original slice and T is the flattened one. E, F, G, and H are the coefficients, which will, in many cases, be constants known at compilation time. Then

Fig. 11.1

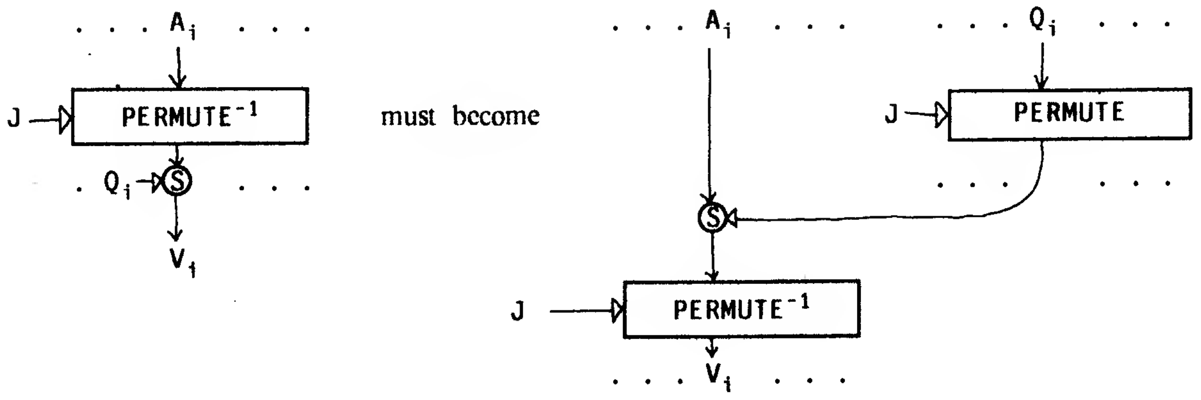


11.3 Getting Permuters and Conditionals Out of the Way

A few things may cause the original array slice not to go through a series of **select** operators in the simple way shown above. Foremost of these is a permuter. The situation is corrected by moving all **permute** operators "downstream" past all **select** operators. This will cause all multidimensional array slices to go through only selects until a scalar result is obtained before going into a permuter. It has the added effect of making permuters handle only scalar data. This may be useful in two regards: It avoids reference accounting problems that could arise from manipulation of array values, and it avoids the possibility of sending array values from one part of the computer to another in an unpredictable manner. These points will be discussed in Chapter 15.

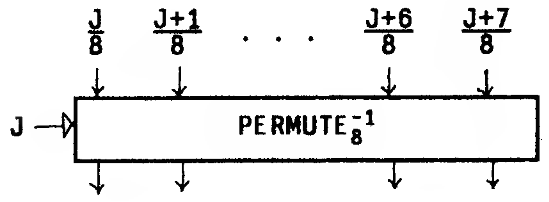
When a **permute** operator is moved downstream past a row of **select** operators, the index values for the selects need to be permuted to compensate. Specifically,

Fig. 11.2



This is not always an inconvenience. In many cases the index values going into the selects (the Q_i values) are all the same, so no permutation is needed. In other cases the Q_i values are related to each other in a simple way whose interaction with the permuter can be predicted. For example, this:

Fig. 11.3



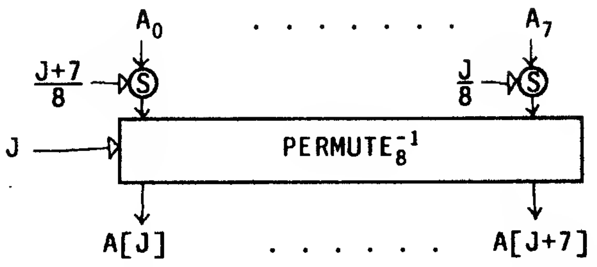
is equivalent to

Fig. 11.4

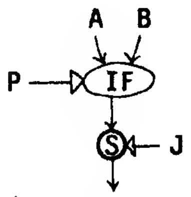
$$\frac{J+7}{8} \quad \frac{J+6}{8} \quad \cdots \quad \frac{J+1}{8} \quad \frac{J}{8}$$

So the standard multi-select of figure 9.4 becomes

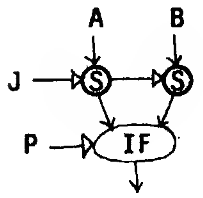
Fig. 11.5



Another thing that needs to be moved downstream past any **select** operator is the conditional. Whenever we have



we turn it into



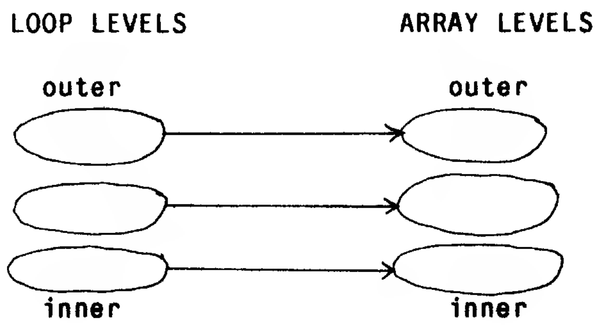
How easy it is in practice to do this in reasonably general cases remains to be seen.

If these things are done, and no other pathological program structures conspire to make life difficult, K-dimensional array slices should go only to K consecutive **select** operators. When this is so, the slices can be flattened.

11.4 Creating Flattened Slices

Multidimensional arrays of the sort we are interested in are created by nested loops in which each loop level creates one dimension of the array. Using the nesting diagrams of Chapter 5, we require strict correspondence with no crossing of arrows.

Fig. 11.6



At each loop level, the elements are assembled, one per cycle, into a linear array. Those elements are either scalars or lower level arrays produced by inner loops. We will assume that reference interval for creation is always one. If it were greater, the array would be created with holes in it. If less, elements would be overwritten. A compiler could, in principle, turn these cases into the equivalent cases with reference interval equal to one.

Now the fact that there is a strict correspondence between loop levels and array levels may be just a shortcoming of the language we are using. It is certainly possible, in FORTRAN, to write

```
DO 99 I = 1, 10
DO 99 J = 1, 10
DO 99 K = 1, 10
DO 99 L = 1, 10
. . . .
A(K, L, J, I) = expression
```

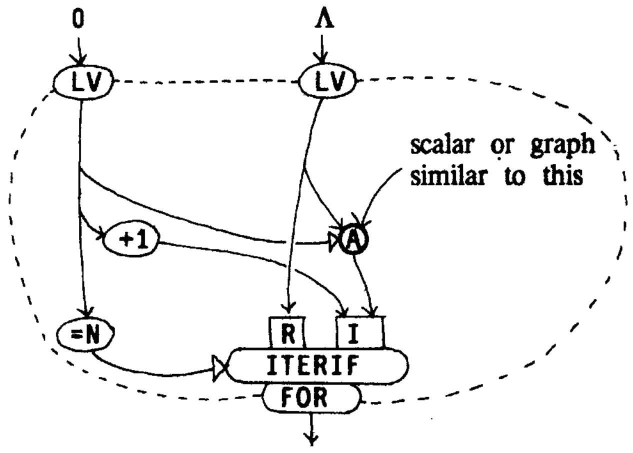
That is, a FORTRAN or other statement-oriented language can fill a multidimensional array with any loop level/array level correspondence. To do the same in VAL is unnatural because of the applicative nature of the language and the fact that it uses the "vector of vectors" model for multidimensional arrays. One may well be able to design a language that can easily define arrays with any index order. It is not necessary, however. One could just reorder the indices to match the loop nesting structure and then use the straightforward VAL notation. Hence we will assume the array indices correspond with the loop levels exactly.

The loop at each level looks something like this:

```
for I, A := 0, Λ
do  if I=N then A
    else
        let X := . . . either a scalar or the array
                       produced by a loop similar to this
        in  iter I, A := I+1, A[I: X] enditer
        endlet
    endif
endfor
```

The graph to produce this is

Fig. 11.7

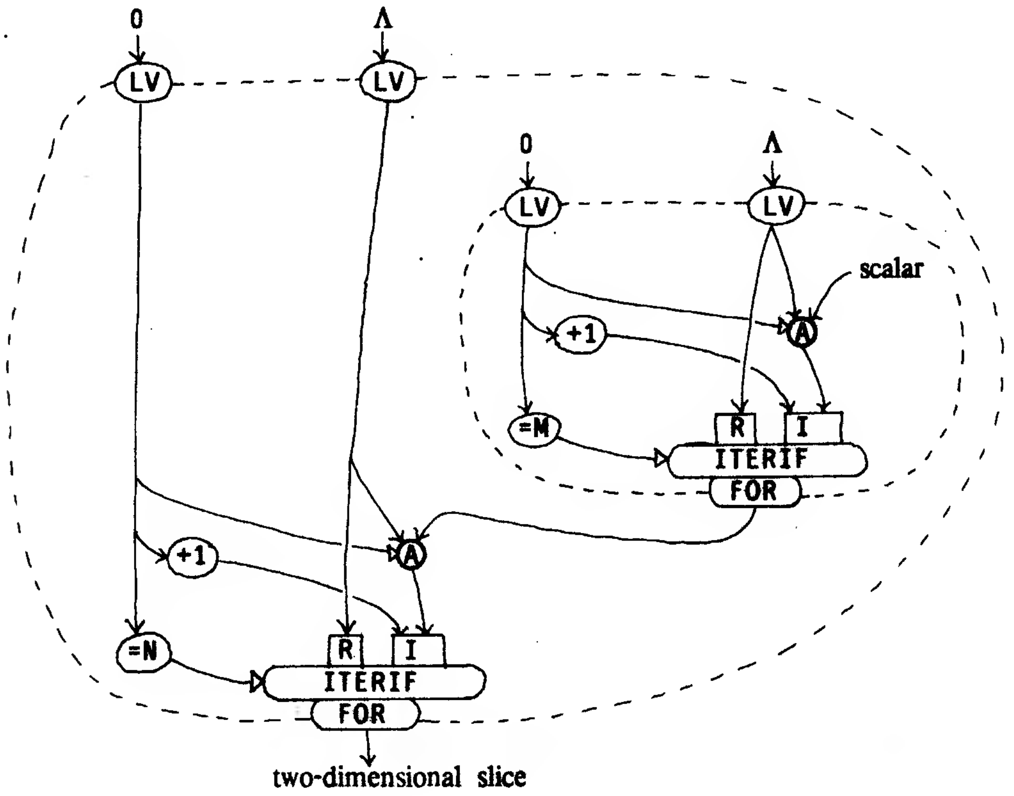


These graphs are of course nested.

Even if loops are unfolded by arbitrary amounts at arbitrary levels, each instantiation looks like this, as long as the index variable is rescaled and the amount of unfolding is equal to the array interlace at each level. We will ignore for now the possibility that the interlace might not equal the unfolding. Each slice is produced by a graph such as the one above.

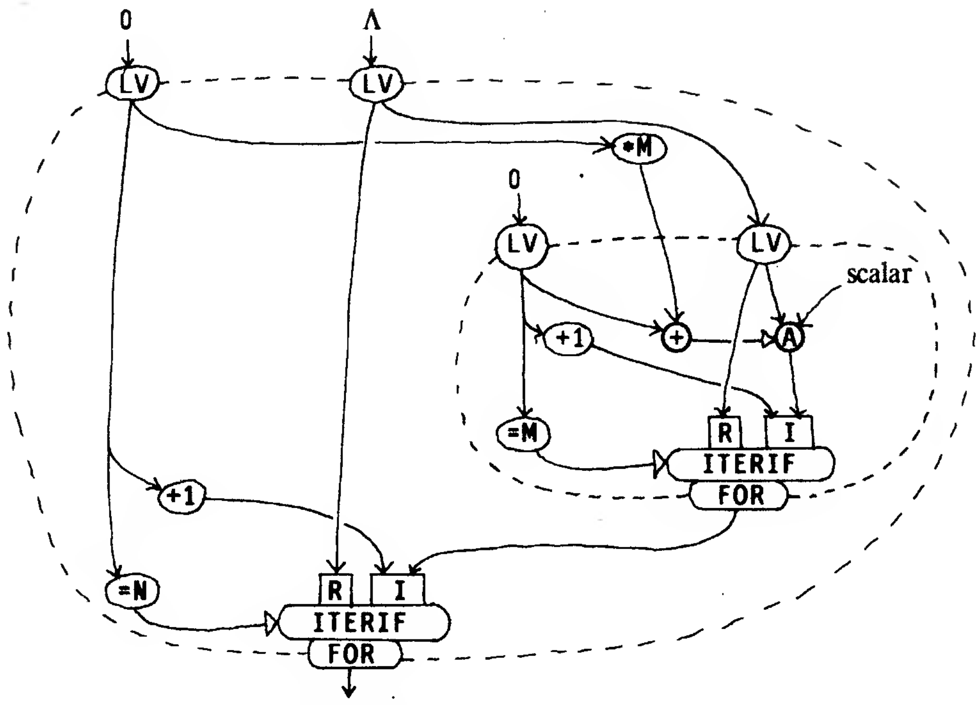
Now it happens that, if the multidimensional array slice is flattened with the last index varying most rapidly, the order in which the scalar values are produced by the innermost loop is the order in which they are placed in the flattened slice. Instead of having the innermost loop create separate rows which the outer loops assemble into the final array slice, only one linear array is used. The innermost loop simply adds its elements to that array value. The outer loops pass the result of one inner loop into the next inner loop. Graphically, we could flatten a two dimensional array slice produced by this loop

Fig. 11.8



by having it produced instead by this

Fig. 11.9



flattened slice

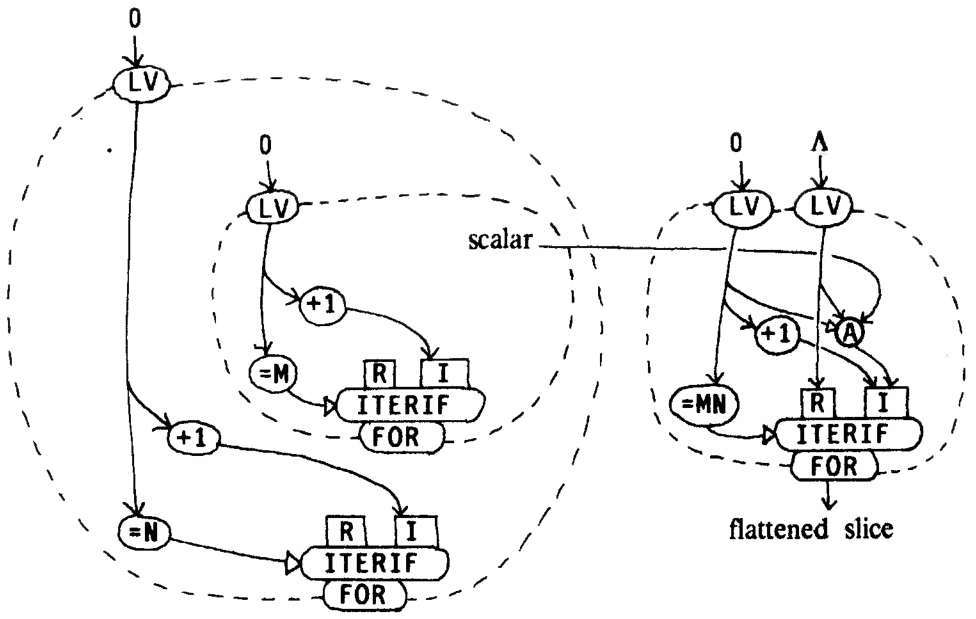
Now the array slice is one-dimensional. Elements are appended at only one level. By passing the slice from one inner loop to the next, the "rows" that the inner loop would normally produce are concatenated end to end, as the flattening operation requires. This construction can be used for any number of dimensions.

11.5 Flattening the Control Structure

The part of the above graph that produces the sequence of scalar values can be separated from the part that assembles them into the flattened array slice. The latter part can easily be seen to function by simply appending the scalars into the slice in linear order. It clearly doesn't need to be a nested loop -- it can be "flattened" into a simple loop.

We now have

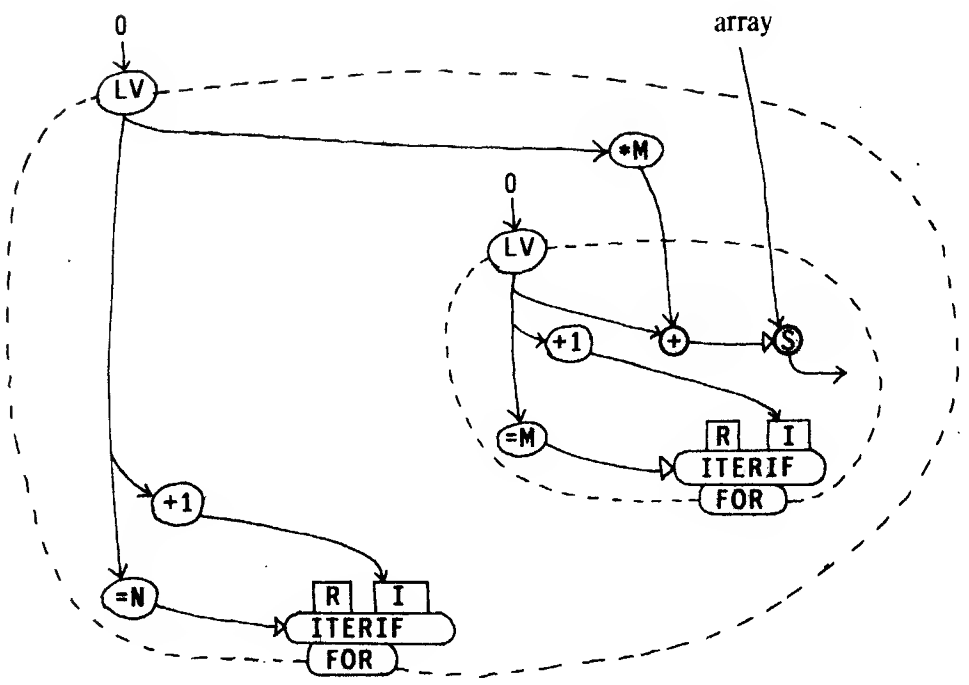
Fig. 11.10



The graph on the right is an "array packer". It is a structure that the hardware should be able to handle very efficiently. We will return to this structure in Chapter 15.

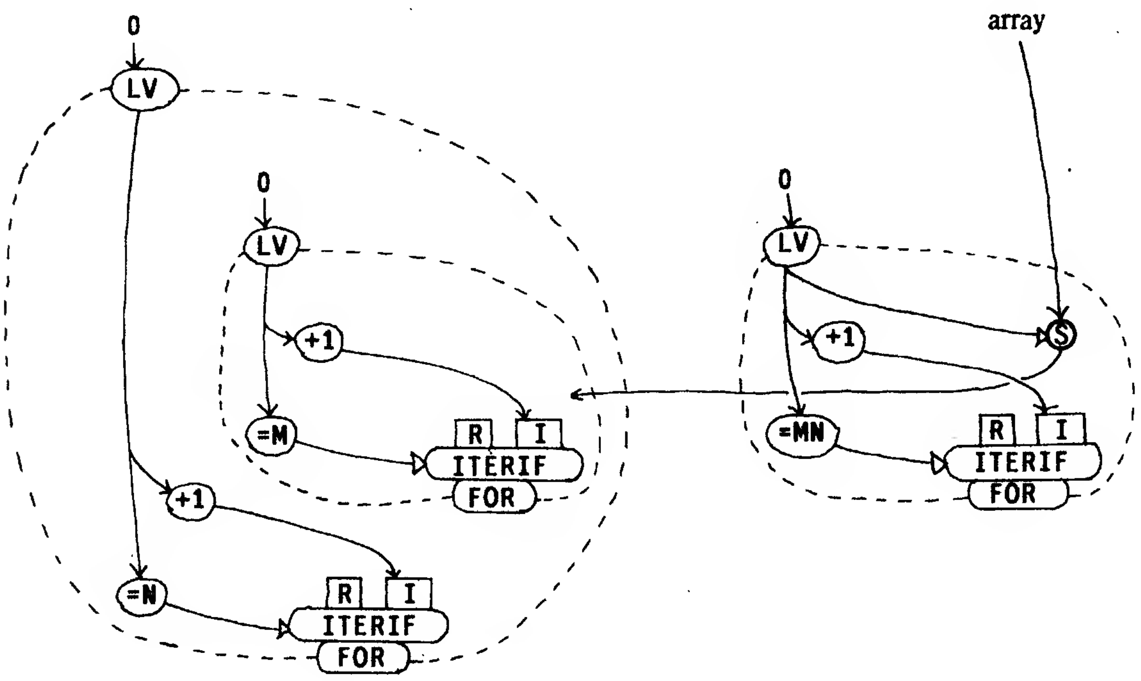
A special graph structure can also be used for select operators, if the loop nesting matches the array structure and the reference interval is one at each level. A nested loop that refers to A[I][J], where I is the outer loop index variable and J the inner one, might look like this after array flattening:

Fig. 11.11



This can be turned into

Fig. 11.12



The graph on the right is an "array unpacker". It is also a structure that the hardware ought to be able to handle efficiently.

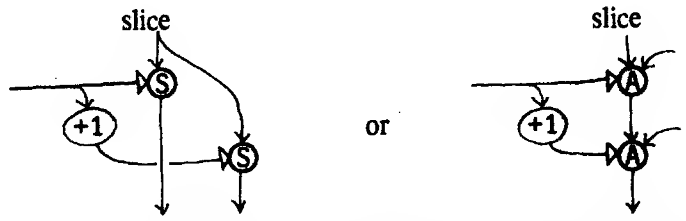
Array packers and unpackers have an interesting property that make them potentially able to save enormous amounts of memory space. In certain cases, packer/unpacker pairs can be removed, and an array can be passed from one loop to another as a stream of scalar values. This will be discussed in Chapter 14.

11.6 Interlace Not Equal to Unfolding -- Alternators

If the interlace is smaller than the amount of loop unfolding, we know that each slice must go to multiple select or **append** operators, each in a different loop instance.

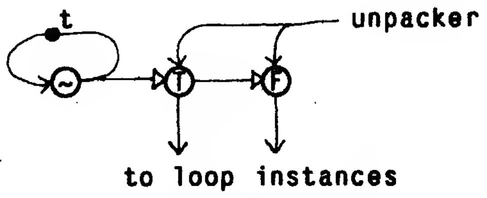
We either have

Fig. 11.13



Using an array unpacker, we have to send the unpacked elements to the separate instances in interleaved order:

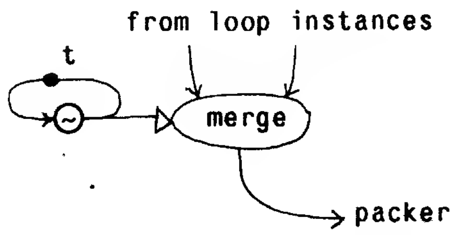
Fig. 11.14



This subgraph can be considered a one-input two-output "alternator". The alternating true and false boolean tokens can be obtained in a variety of other ways, such as using the low order bit of the index in the unpacker. In any case, careful attention needs to be paid to boundary conditions to make sure that this starts up in the correct state.

Using an array packer, we need a two-input one-output alternator, like this:

Fig. 11.15



When the ratio of the loop unfolding to array interlace is greater than two, more complex alternators are required.

If the interlace is larger than the amount of loop unfolding, we have multiple array packers or unpackers connected to the same loop instance. This problem is also solved with appropriately connected alternators. These alternators perform a function equivalent to that of the permuters that were used to solve the same problem in Chapter 4.

In multiple dimensions, when the interlace at the various levels does not match the unfolding at the corresponding level, the data must go through several layers of alternators, each one designed to correct the mismatch at that level. The actual structure of such networks can be quite complex.

12. BOUNDARY CONDITIONS

The loops considered so far have all been completely uniform in their use of arrays. That is, the correspondence between loop cycle number and array index has been "affine":

$$\text{array index} = \alpha \cdot \text{cycle number} + \beta$$

where α and β may or may not be known prior to execution, but are known not to change once the loop begins.

The results can easily be extended to loops that refer to arrays in a "piecewise-affine" way. A piecewise-affine correspondence is one that looks like this:

$$\text{array index} = \begin{array}{ll} \alpha_1 \cdot \text{cycle number} + \beta_1, & \text{for} \quad \gamma_1 \leq \text{cycle number} < \gamma_2 \\ \alpha_2 \cdot \text{cycle number} + \beta_2, & \text{for} \quad \gamma_2 \leq \text{cycle number} < \gamma_3 \\ \quad \cdots & \\ \alpha_k \cdot \text{cycle number} + \beta_k, & \text{for} \quad \gamma_k \leq \text{cycle number} < \gamma_{k+1} \end{array}$$

It is important that the number of pieces (k in the formula above) be fairly small. This is because the size of the program itself may increase in proportion to k. Recall from Section 1.4 that we are dealing with arrays much larger than the program size that we could tolerate. It follows that the number of pieces must be very small in comparison to the number of elements in the array.

The solution is fairly straightforward in principle (though, like so many things in this report, tedious in its details): Analyze the array accesses separately for each piece, and generate the appropriate code. Use conditionals to select the correct code in each piece. For example, something like

```
%% I = the loop index variable
J := if I < 50 then 2*I+47 else 4*I endif
. . . . .
. .  A[J]  . .
. . . . .
```

gets translated to

```
. . . . . .
A1 := A[2*I+47]
A2 := A[4*I]
. . . . . .
. .  if I < 50 then A1 else A2 endif
. . . . . .
```

Now, since I is the loop index variable, the appropriate code for A[2*I+47] and A[4*I] may be easily generated.

Of course, if some of the pieces cover only one loop cycle, as is often the case with boundary conditions, the compiler should recognize that the index variable is a "constant" in the array access for that piece, and make the appropriate simplification.

The extension of this to nested loops is straightforward.

12.1 Dividing Loops

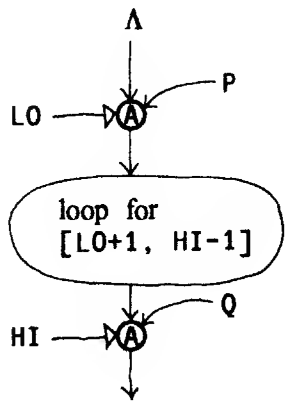
Another technique that is useful in some cases is to divide the loop into a series of loops, one for each piece, which are then concatenated to form the original one. Each loop in the series processes only the cycles of the original loop that correspond to its piece, and passes its final loop variables to the next loop. Within each loop, array references are purely affine.

This technique is useful when the computations involving the various pieces differ significantly, since the amount of code that is shared among the pieces is small. Boundary conditions provide a typical example of this. If we have a loop such as

```
forall I in [LO, HI]
construct
    if I=LO then P
    elseif I=HI then Q
    else f(I)
    endif
endall
```

It would be appropriate to translate it as

Fig. 12.1



Of course, the programmer could have specified this decomposition in the source program, writing something like

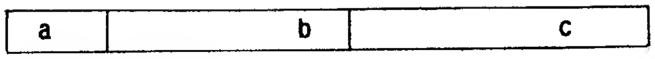
```
let X := forall I in [LO+1, HI-1] construct f(I) endall
in  X[LO: P][HI: Q]
endlet
```

but the representation as a single **forall** with a conditional inside seems to be the preferred programming style in VAL (largely because the explicit decomposition is hard to express with nested **forall**'s). For this reason, automatic division of loops is a useful optimization technique.

12.2 Dividing Nested Loops with Multiple Array References

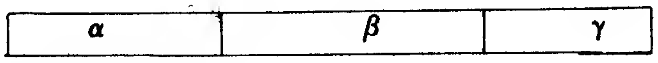
When several array references (to the same or different arrays) are involved, each of which is in several affine pieces, it is necessary to divide the loop wherever any of the array references requires it. If one array reference requires division like this

Fig. 12.2



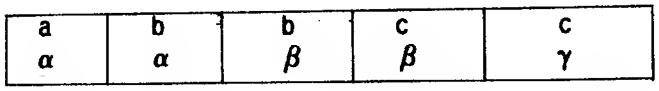
and another requires division like this

Fig. 12.3



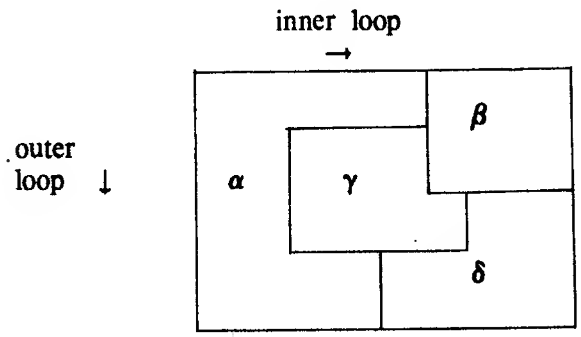
the loop must actually be divided like this

Fig. 12.4



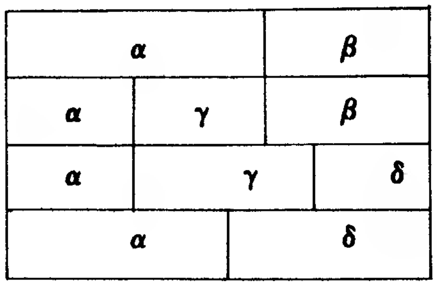
If we have nested loops, and various regions must be divided up so that all references will be affine in each piece, we must choose a fine enough mesh in each direction. For example, if we have a double loop with these pieces

Fig. 12.5



The outer loop must be divided into four pieces, and then the inner loops independently divided like this on those pieces:

Fig. 12.6



13. LARGE SCALE PROGRAM STRUCTURE

This chapter will examine the applicability of the techniques of array interlace and loop unfolding to programs in general.

Any program whose array references are all piecewise affine can be treated by the methods of the preceding chapters. This needs to be taken in perspective: It does *not* mean that arbitrarily large amounts of parallelism can be found, for there may be genuine bottlenecks in the algorithm. It *does* mean that bottlenecks arising from array references can be removed. That is, we can remove the apparent bottleneck that arises from the fact that repetitive array operations seem to pass a single array token from one operation to another in sequence. To the extent that there is parallelism in the algorithm, the array operations can be restructured so that they will not prevent that parallelism from being exploited.

Programs consist of operators, conditionals, and iteration loops, put together in an acyclic data flow graph. (**Forall** loops are considered to be a special case of iterations.) In the case of conditionals and iterations, there is a hierarchical structure -- the subgraphs of conditionals and iterations are themselves acyclic data flow graphs composed of operators, conditionals, and iterations. (The inner mechanism of an iteration involves cyclic flow of tokens, but, in an abstract structural sense, programs are built out of a hierarchy of acyclic graphs of operators, conditionals, and iterations.)

Every array reference is inside some (possibly empty) nested collection of loops. The index expression for the reference (or, if the array is multidimensional, each index expression) may depend on one or more of the loop index variables. Now there are several ways in which an index expression in an array reference might be badly behaved.

1 It might not depend *solely* on loop index variables -- there might be contributions from other calculations that are unpredictable or "random".

2 The dependence might not be affine.

3 The loop variables might not have a uniform increment from one loop cycle to the next.

Collectively, these conditions mean that the reference interval is not well defined.

Now a failure in one or more of these conditions might not be fatal. If the failure occurs just at the boundaries or, in any case, at a very small number of points, we can take those points out of the loop, as discussed in Chapter 12. The important point is that the vast majority of references depend solely on loop indices with known reference interval. A few occasional references that don't satisfy these criteria won't matter.

Another failure that is not fatal is a dependence on something that is not a loop index variable with known reference interval, but is nevertheless known to be a multiple of the interlace. This situation arose in the periodic cyclic reduction algorithm of Chapter 4. It is why an initial expansion was performed on the outer loop.

The important point is this: If the index expression of an array reference depends solely on loop variables with known reference interval, and appropriate amounts of array interlace and loop unfolding are used, that array reference will not limit the parallelism of those loops. That is, no sequential data dependency will exist in those loops as a consequence of that array reference. This does not mean that the loops can really be successfully unfolded -- there might be *other* sequential data dependencies.

Not all parts of a program can necessarily benefit from this. Some parts are not in loops (or, because they are boundary conditions, must be taken out of their loops), or are in loops that do not unfold well.

Not all parts of a program need to benefit from this. The overall performance of a program is usually determined by one critical part -- usually an innermost loop. The performance of a program on a large applicative computer will depend on whether loop unfolding and array interlace can be performed on the critical parts. If one is very successful at this, some of the parts that had not previously been critical might become critical. In the limit of extreme loop unfolding, bottlenecks might arise in unexpected places.

13.1 Example -- the Cyclic Reduction Algorithm

The complete cyclic reduction algorithm follows [36, 44]. This solves linear systems of equations in which the nonzero elements in the coefficient matrix are permitted only along the main diagonal or the diagonals immediately above and below it.

The traditional algorithm for solving this problem, sometimes called "LU decomposition", has sequential data dependencies whose length is of order `N`, the number of equations in the system. This makes it unsuitable for vector or array machines. The cyclic reduction algorithm overcomes this problem. It has roughly the same number of arithmetic operations as LU decomposition, but much larger numbers of them can be executed in parallel. The longest chain of sequential data dependency is of order `log(N)`. This property of having the theoretical absolute minimum computation time much shorter than the execution time on a sequential machine makes cyclic reduction suitable for data flow computation as well.

The algorithm presented here is very similar to the "periodic cyclic reduction" algorithm presented in Section 4.9. The earlier algorithm was a specialization of this one for solving Poisson's equation, with some extra computation (the manipulation of the scalar "`A`") to satisfy periodic boundary conditions. The earlier algorithm manipulated one array: `Q`. This one manipulates four arrays: `A`, `B`, `C`, and `D`. It also does more arithmetic -- the computation of `MU`, `LAM`, and `RHO`.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%% Solve a tridiagonal system of equations.
%%%
%%% N is required to be a power of 2.  The incoming vectors
%%%   are:
%%%     M[1..N-1] - the diagonal elements of the matrix.
%%%     U[1..N-2] - the above-diagonal elements
%%%     L[2..N-1] - the below-diagonal elements
%%%     R[1..N-1] - the right-hand side.
%%%
%%% This returns the vector [1..N-1], call it X, such that
%%%
%%%     M[1] U[1]  0    0    0  . . .     0         X[1]           R[1]
%%%     L[2] M[2] U[2]  0    0  . . .     0         X[2]           R[2]
%%%      0   L[3] M[3] U[3]  0  . . .     0     *   X[3]     =     R[3]
%%%        . . . . . . . . . . . . . . .            ....           ....
%%%      0    0    0 .. L[N-2] M[N-2] U[N-2]        X[N-2]         R[N-2]
%%%      0    0    0 ..    0   L[N-1] M[N-1]        X[N-1]         R[N-1]
%%%
%%% This uses the "cyclic reduction" algorithm.
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function solve(M, U, L, R: areal ; N: integer returns areal)
type areal=array[real]

%%% REDUCTION

let
   AZ, BZ, CZ, DZ :=
   for IH, AA, BB, CC, DD := 1, M, U, L, R
   do
      if IH=N/2 then AA, BB, CC, DD
      else
         let
            ID := IH*2 ;
            NEWAA, NEWBB, NEWCC, NEWRR :=
            for J, A, B, C, D := ID, AA, BB, CC, DD
            do
               if J=N then A, B, C, D
               else
                  let
                     MU := AA[J-IH]*AA[J+IH] ;
                     LAM := CC[J]*AA[J+IH] ;
                     RHO := BB[J]*AA[J-IH] ;
                  in
                     iter J, A, B, C, D :=
                        J+ID,
                        A[J: LAM*BB[J-IH]+RHO*CC[J+IH]-MU*AA[J]],
                        B[J: RHO*BB[J+IH]],
                        C[J: LAM*CC[J-IH]],
                        D[J: LAM*DD[J-IH]+RHO*DD[J+IH]-MU*DD[J]]
                     enditer
                  endlet
               endif
            endfor ;
         in
            iter IH, AA, BB, CC, DD :=
               ID, NEWAA, NEWBB, NEWCC, NEWRR
            enditer
         endlet
      endif
   endfor ;
```

```
%%% SUBSTITUTION

in
   for ID, X := N, DZ
   do
      if ID=1 then X
      else
         let
            IH := ID/2 ;
            NX :=
            for J, NNX := IH, X
            do
               if J=N+IH then NNX
               else
                  let
                     ALPHA := if J=IH then 0.0 else CZ[J]*X[J-IH] endif ;
                     BETA := if J = N-IH then 0.0 else BZ[J]*X[J+IH] endif ;
                  in
                     iter J, NNX :=
                        J+ID, NNX[J: (X[J]-ALPHA-BETA)/AZ[J]]
                     enditer
                  endlet
               endif
            endfor ;
         in
            iter ID, X := IH, NX enditer
         endlet
      endif
   endfor
endlet
endfun
```

The same analysis that was performed in Section 4.9 applies here. The manipulations of arrays A, B, C, and D are virtually identical to the manipulation of Q in Section 4.9. As in Section 4.9, if we use a 16 way interlace on A, B, C, and D, it is appropriate to perform an initial unfolding of 4 on the outer reduction loop (for values of IH = 1, 2, 4, and 8) and to use various amounts of unfolding on the inner loops. In the substitution part, it is appropriate to perform a final unfolding of 4 on the outer loop, and to use various amounts of unfolding on the inner substitution loops.

13.2 The Fast Fourier Transform

The following program is the complex discrete Fourier transform over `N` points, computed by the Cooley-Tukey ("fast Fourier transform") algorithm [15]. `COSTAB` is the table of sines and cosines. The compiler is expected to remove the apparent record structure of data of "complex" type. All arrays of complex type (e.g. `A` or `AL`) are actually two arrays, one carrying the real parts and one the imaginary parts.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%% Complex Discrete Fourier Transform over N points.
%%% N must be a power of 2, and at least 4.
%%% The argument and result are defined on [0..N-1].
%%% INVERT tells whether to invert the transform.
%%% The result is not normalized: it should be divided by SQRT(N)
%%%   to get the true transform.

%%% If INVERT is false, this returns RESULT such that
%%%                n-1
%%%   RESULT[K] = SUM exp(2*pi*i*j*k/N) ARG[j]  for K in [0, N-1]
%%%                j=0
%%% If INVERT is true, the argument to exp is negated.

%% This requires 3 quadrants of cosines in COSTAB, defined over
%%   [ -N/4+1, N/2-1], with COSTAB[j] = COS(2*pi*j/N).

function DFT(ARG: array[complex] ; COSTAB: array[real] ; N: integer ;
   INVERT: boolean returns array[complex])

type complex = record[re, im: real] ;

for Z, A := 1, ARG ;
do
   if Z = N then A
   else
      let
         NEWZ := Z*2 ;
         AL, AH :=
         forall L in [0, N/NEWZ-1], J in [0, Z-1]
            P := A[Z*L+J] ;
            Q := A[Z*L+J+N/2] ;
            S0 := COSTAB[N/4-J*N/NEWZ] ;
            S := if INVERT then -S0 else S0 endif ;
            C := COSTAB[J*N/NEWZ] ;
            QWre := Q.re*C-Q.im*S ;
            QWim := Q.re*S+Q.im*C ;
         construct
            record[re: P.re+QWre ; im: P.im+QWim],
            record[re: P.re-QWre ; im: P.im-QWim]
         endall ;
         NEWA := forall K in [0, N-1]
         construct
            if mod(K, NEWZ) < Z then AL[K/NEWZ][mod(K, Z)]
            else AH[K/NEWZ][mod(K, Z)]
            endif
         endall ;
      in
         iter A, Z := NEWA, NEWZ enditer
      endlet
   endif
endfor
endfun
```

If we assume that the input and output arrays have 8 way interlace, it is useful to perform an initial unfolding of 3 on the outer loop, since the optimal structure of the inner loops is different for Z = 1, 2, and 4. The interlace and unfolding of the **forall** loops are as follows:

First outer cycle, Z = 1:

A interlace = 8, size of each slice = $\frac{N}{8}$

AL, AR interlace = 4 * 1, size of each slice = $\frac{N}{8} * 1$

Since size of slice in second index is one, it is not really an array, and the slices can be considered to be one-dimensional.

Unfolding of **forall** creating AL, AR is 8 * 1

Since the inner loop has only one cycle after expansion, it is removed.

Unfolding of **forall** creating NEWA is 8

Second outer cycle, Z = 2:

A interlace = 8, size of each slice = $\frac{N}{8}$

AL, AR interlace = 2 * 2, size of each slice = $\frac{N}{8} * 1$

Since size of slice in second index is one, it is not really an array, and the slices can be considered to be one-dimensional.

Unfolding of **forall** creating AL, AR is 4 * 2

Since the inner loop has only one cycle after expansion, it is removed.

Unfolding of **forall** creating NEWA is 8

Third outer cycle, Z = 4:

A interlace = 8, size of each slice = $\frac{N}{8}$

AL, AR interlace = 1 * 4, size of each slice = $\frac{N}{8} * 1$

Since size of slice in second index is one, it is not really an array, and the slices can be considered to be one-dimensional.

Unfolding of **forall** creating AL, AR is 2 * 4

Since the inner loop has only one cycle after expansion, it is removed.

Unfolding of **forall** creating NEWA is 8

Later outer cycles, Z $\geq$ 8:

A interlace = 8, size of each slice = $\frac{N}{8}$

AL, AR interlace = 1 * 8, size of each slice = $\frac{N}{2Z} * \frac{Z}{8}$

Unfolding of **forall** creating AL, AR is 1 * 8

Unfolding of **forall** creating NEWA is 8

AL and AR slices, being two-dimensional, are flattened.

No permuters are required anywhere, and the worst case of an array slice being used in more than one operation occurs when Z is 1, 2, or 4: In the **forall** that creates AL and AR, those array slices go through two **append**'s in each cycle.

The remaining array reference that needs to be considered is the **select** reference to COSTAB. If COSTAB has an interlace of 8, no permuters will be needed, but, except for the last few cycles of the outer loop, only slice zero will be used. Since the **forall** that makes reference to COSTAB is unfolded 8 ways, slice zero needs to go to 8 **select** operators. The bottleneck implied by this can be removed by using 8 copies of the slice, one for each unfolded loop instance. Since COSTAB is a constant array, it is easy to provide multiple copies.

14. PIPELINING AND ARRAY REMOVAL

It is very common in an iteration loop to have the computation of the termination condition not depend on the outcome of the bulk of the computation, and the data paths from any value of any loop variable to its next value be very short in comparison to the bulk of the computation. When this happens, a very useful form of "decoupling" may be possible.

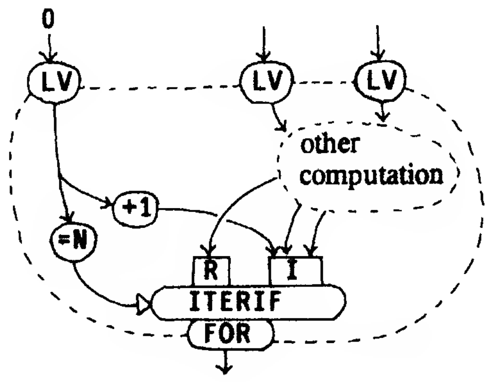
The first criterion is satisfied by, among other things, any loop that is controlled by a counter, such as

```
for I, X, Y := 0, <other initial values>
do  if I=N then <values to return>
    else iter I, X, Y := I+1, <other new values> enditer
    endif
endfor
```

Any forall loop is clearly of this form, for example.

The graph for this is

Fig. 14.1

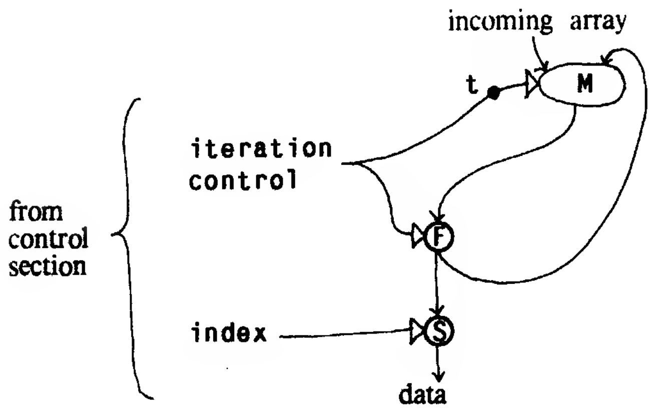


The significance of this situation is that the flow of tokens through the control operators can proceed far ahead of the rest of the computation. The control computation for the entire loop could complete before the bulk of the graph completed its first cycle. The control part of the loop can "decouple" from the rest of the loop, in the same manner as was discussed in Chapter 10. As in that chapter, FIFO buffers are required to hold the boolean tokens that have been created by the control part but not yet consumed by the bulk of the computation.

The other criterion is that, although some computation paths may be long, the cyclic paths through which loop variables must pass are each relatively short. That is, the dependence of a "next" value of a loop variable on the "present" value of *that same variable* is simple, although its dependence on the present values of *other* loop variables might involve a lengthy computation.

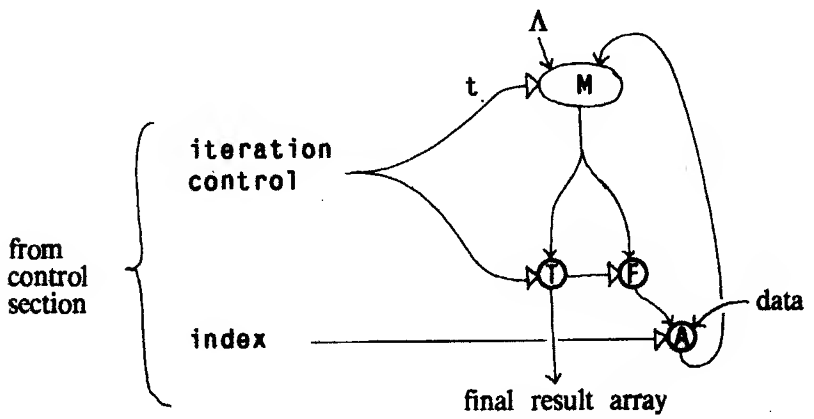
The prototypical examples of such loop variables are the array unpacker:

Fig. 14.2



and the array packer:

Fig. 14.3



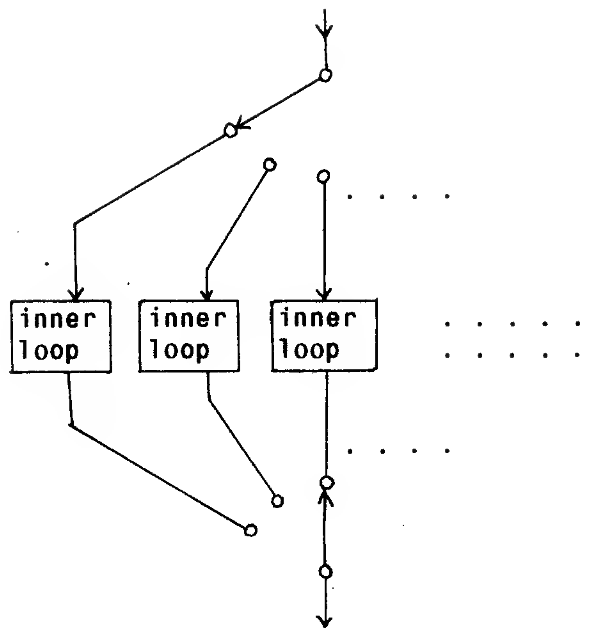
If these two (or any other subgraphs with short cycle paths) are connected so that data from the unpacker goes through a lengthy computation and then to the packer, it may be possible for the unpacker to run many cycles ahead of the packer. The control section must be independent and able to run at least ahead of the unpacker, and there must at least be a FIFO buffer to store the control tokens that the packer has not yet consumed. The results of the extra unpacks can then be *pipelined* through the main body of the computation. That is, several "waves" of tokens can be in transit through the graph simultaneously. The acknowledge rules of the static data flow computer will maintain an orderly flow. This pipelining can improve the throughput/instruction space ratio of the system tremendously.

The capability of the graph to allow this pipelining is heavily dependent on many aspects of the graph's structure. Programs that support this pipelining are referred to as "pipe structured" and are examined in [25]. The ability to support pipelining can often be enhanced by inserting null operations in strategic places, an act known as "balancing". This is examined in [32].

To support pipelining, the graph must have the property that no operator or subgraph may have a long latency (time that elapses from the instant a token enters it until the corresponding result emerges) unless that operator or subgraph itself supports pipelining. Individual operators have short latency, but subgraphs that are iterations typically do not, and iteration loops, viewed from the outside, do not support pipelining. That is, a loop of one hundred cycles cannot accept a second wave of input tokens to start its second hundred cycles until it has at least started the hundredth cycle from the first wave. Any loop nested inside the outer loop in question would therefore appear to render pipelining of the outer loop impossible.

This problem can be overcome by interleaving the entire inner loop. Suppose the inner loop has a latency of 80 microseconds, but we want a new cycle of the outer loop to start every 10 microseconds. A token must pass each point of the outer body every 10 microseconds, but a token can enter a copy of the inner loop only every 80 microseconds. We can use 8 copies of the inner loop (expensive, but we get the performance that we pay for) whose inputs and outputs are connected through alternators. Alternators were described in Chapter 11. The alternators can be visualized as 8 position rotary switches:

Fig. 14.4



Every incoming token goes through the upper switch into one of the inner loops, and that switch advances to its next position. When the lower switch receives a token along its selected line, it passes that token along and advances to its next position.

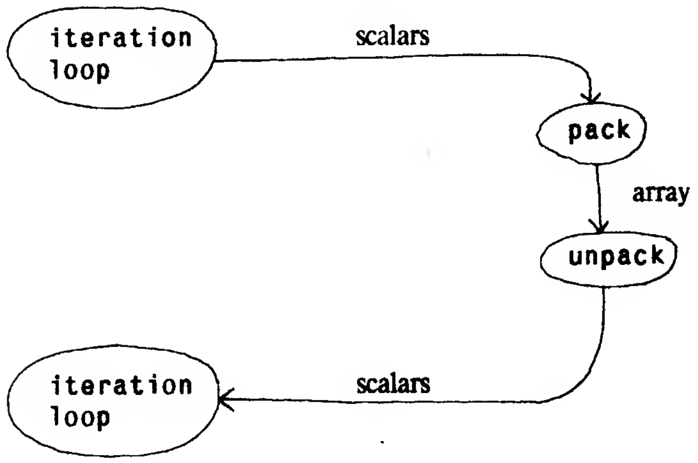
Of course we want all instances of the inner loop to be as efficient as possible, which may require pipelining them, as well as the usual unfolding.

A nice property of the pipelining that can occur in a data flow computer is that it doesn't need to be worked out precisely in advance the way it does in "systolic" systems [39]. If FIFO buffers of approximately correct size are put into the correct places, pipelining will happen naturally. Minor mismatches in the running speeds of the various parts will not cause serious problems. If some transient irregularity occurs (an arithmetic exception may occur somewhere and need to be corrected) the natural load balancing ability of a computer with many independent processors will smooth the computation out.

14.1 Array Removal

It may happen that array packer/unpacker pairs can "cancel" each other. If one is diligent and fortunate in putting all array references in the form of pack and unpack modules, then arrays will only be created by packers, and they will only be consumed by unpackers. The two modules will be found only in pairs, with an array token passing from each packer to its unpacker. The pairs will look like this:

Fig. 14.5



The packer came from the upper iteration, and its control section was extracted from that of the iteration. Likewise, the unpacker came from the lower iteration. Now, if the unpacker expects an array of the same size as the one that was created, the packer/unpacker pair are just a conduit for scalar tokens. That conduit has memory (obviously!) -- it is able to absorb all of the incoming tokens from the upper loop before the lower loop is ready to accept any tokens.

Under what circumstances is this memory unnecessary? If there is some other data dependency from the final result of the upper loop to the start of the lower loop, then the entire array must be stored before the first element is unpacked. In this case the memory is clearly necessary. However, if there is no such dependence, the two loops can cycle simultaneously, and the pack/unpack pair can be eliminated. The control structures of the two loops might even be coalesced into one. The question of when two loops can be made to run in step with each other is the same question that arose in Section 8.2.

One might ask: What kind of program would have two loops that could have been coalesced but run sequentially instead? Why would anyone write

```
    Y := forall I in [LO, HI] construct A[I]+B[I] endall ;
    Z := forall I in [LO, HI] construct Y[I]/Q endall ;
```

when

```
    Z := forall I in [LO, HI] construct (A[I]+B[I])/Q endall ;
```

would do just as well? The answer lies in the realm of software engineering. It may be that using two loops, creating an "obviously unnecessary" intermediate array and then immediately consuming it, can make programs easier to understand in some cases. Programmers presently know from experience that such intermediate arrays are wasteful, and they carefully avoid such waste.

The excess arrays are only wasteful in conventional systems that cannot remove them by optimization. When systems are used for which there is no efficiency penalty for different ways of expressing an algorithm, perhaps algorithms will be expressed in ways that are easier to understand.

15. CONSIDERATIONS OF MACHINE DESIGN

This chapter will discuss a few points relating to the design of an efficient instruction set for a data flow computer. The design issues in a data flow computer are of course complex and diverse. Most of them are far beyond the scope of this thesis. The present discussion will be restricted to just those issues that relate to efficient processing of arrays, and a brief point about pipelines for boolean values.

In the following, "array" means array slice. That is, we are discussing arrays as seen by the hardware itself, as opposed to the arrays that appeared in the original source program.

A generality/efficiency trade-off is pervasive in the handling of arrays. In the most general case, arrays are dynamic, that is, their size can increase unpredictably during the computation. The array "tokens" can be duplicated or destroyed unpredictably, so their reference counts (see below) can take on arbitrary values. **Append** operations can be performed when the reference count is more than one, which requires that the underlying mechanism copy the entire array. (The necessity for this will be discussed below.) Finally, array tokens can be sent through permuters and other control operators to arbitrarily distant parts of the machine. When such a token then enters a **select** or **append** operator, the place where the operation must be performed may be far removed from the place where the array is stored, which leads to a great deal of communication through routing networks.

The type of computer that can handle this is complex indeed. Rough designs for such machines have been formulated [1, 3], but they are not efficient enough for numerical supercomputation with existing technology. We will assume here that such uncontrolled situations do not arise in the computations of interest, and will examine the type of machine that can take full advantage of that fact.

15.1 Array Localization and Management

In an applicative system, local processing of many small arrays throughout the system is the ideal way to proceed. A major objective of the preceding chapters was to develop data flow graphs in which arrays are processed in small subgraphs. Ideally, each array token spends its entire lifetime in such a subgraph. This lifetime usually consists of the allocation of a block of memory somewhere, the filling of that block in a fairly standard type of loop containing **append** operators, the reading of that array, usually in another standard loop containing **select** operators, and finally the release of the array's storage when its token disappears inside some control gate. There are many variations of this, but the general goal is to restrict each array token to a localized subgraph during its entire lifetime. The reason for doing this is to allow all **select** and **append** operations to be executed in hardware units that are physically and logically close to the memory unit in which the array is stored.

This local processing of arrays can and should be reflected in the hardware. The subsystems of the machine that handle arrays are replicated many times over. Ideally, each array token spends its lifetime in just one such subsystem. That subsystem contains the memory devices (e.g. RAM's) in which the array is stored. The mapping, performed by the compiler; from the data flow graph onto the machine includes an assignment of each array-handling subgraph onto an array-handling subsystem. It is desirable that array tokens not be allowed to travel very far. In particular, they should not move from one array-handling subsystem to another -- otherwise there will be an enormous amount of traffic in the communication network as data items are fetched in one subsystem for use in another. This is the reason that, in Chapter 11, we attempted to minimize the passage of array tokens through such things as permuters and conditionals.

It is reasonable to assume that the memory blocks to which array tokens correspond are allocated dynamically. That is, each array subsystem has a dynamic memory management mechanism. When a token is created (by the operation that we have denoted in graphs by the symbol A) a suitably sized block is taken from a "free storage pool". When that token is no longer needed its block is returned to the pool. Completely static allocation, in which each array token in the data flow graph is assigned a specific location in the memory of its array subsystem prior to execution, could also be used for a restricted class of programs. This does not seem worthwhile, since the overhead incurred in dynamic allocation should be quite small.

15.2 Reference Counts

There are two common methods for handling memory management: In the garbage collector method, unused blocks of memory are located through a specific procedure, called garbage collection, when the free storage pool becomes empty. This is done by tracing all blocks of memory that are in use. This method can be quite complicated, especially in a machine in which array tokens can pass through routing networks. It is generally used only when memory references can be circular and the simpler reference counting scheme will not work.

In the reference count method, the system knows at all times which blocks are in use, and returns each block to the free storage pool as soon as it ceases to be in use. This information is generally stored in the form of a reference count, which is the number of references to that block. There is a reference count associated with each block. The reference count must be carefully manipulated whenever any operation is performed that changes the number of references. The reference counting method is simpler than garbage collection, but does not work if circular lists can be made. This is because a circular list could be isolated, that is, have no references into it from the rest of the computation, but still have nonzero reference counts because of the mutual references within the list. Such lists cannot be created in an applicative system, so reference counting is the method of choice.

In a data flow graph, each token containing an array value counts as a reference to that array. If arrays can be stored in arrays, each storage of an array in another array also counts as a reference. Since we assume that arrays are flattened, this does not occur, so we need not consider it further. It is not a serious problem in any case.

Every time an array token is duplicated in any operation, such as sending a result from one operator to several destinations, the count must be increased. Every time an array token is destroyed, the count must be decreased. If it goes to zero, its space must be reclaimed. The principal operations that destroy tokens are the "true" and "false" gates used in conditionals and iterations. In fact, it is the gate in an iteration that causes a token to disappear after the last cycle, if that array is not being passed on to some other part of the graph. The other possible operation that destroys an array token is the **select** operator. Since an array token enters but only a scalar leaves, it effectively destroys its reference. As will be seen later, it is useful to redefine this operator so that it preserves its array token.

The reference count plays another very important role in an applicative system. Throughout this thesis we have treated the **append** operator as though it were a simple thing -- no more complex to perform than the "subscripted left-hand-side" array writing operation of conventional systems. This is not really true. In an applicative system, no value, array or otherwise, is ever allowed to "change". Instead, new values are created. The **append** operator must, in general, *copy its entire array argument* and modify only the copy. In an efficient computer this is, of course, atrocious.

The trick that saves the day is that, if the incoming array has reference count equal to one, that array would be discarded and its space reclaimed right after the copy is made. That being so, the new array might as well use the same space. So, if the reference count is one, the **append** operator can write the new data *in situ*, making it as simple as we have been assuming. An applicative operation can be made out of non-applicative actions.

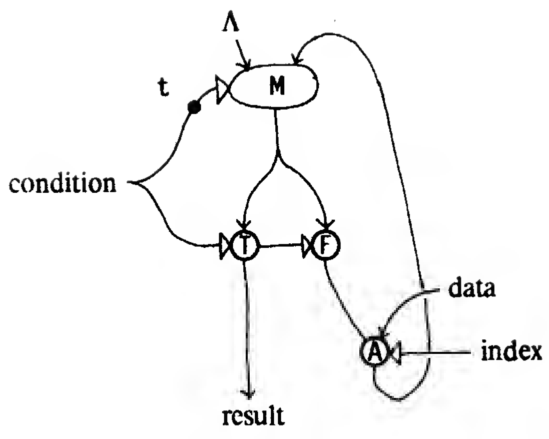
15.3 Pseudo-reference Counts

A straightforward implementation of the reference count scheme would have the count for each array stored in the actual memory along with the array's data. When an array is created (that is, an array token proceeds forth from a Λ node) its count is set to one. All operations through which it can pass increase the count if they send the token to multiple destinations. (In a static data flow graph the number of destinations is an unchanging property of each node.) A "T" or "F" gate decreases the count if it destroys the token, which depends on the value of the boolean control token. A **select** operator always decreases the count. Any time the count is decreased, the result is checked. If it is zero, the array's storage is reclaimed.

All of these reference count manipulations are expensive. The number of memory references to change the count could exceed the number of references to the data. They are also unnecessary, in most cases. With a little extra care, the graph can be constructed so that the reference count of a token at any point in the graph is known to the compiler, before execution begins. In fact, it can be known to be equal to one, which will permit *in situ* **append** operations. By having the compiler compute "pseudo-reference counts", the program doesn't need to compute actual counts during execution. Operations to reclaim arrays can be inserted into the graph at the points that the compiler determines that the count would have gone to zero.

In the simple cases, the reference count is known to be one throughout a loop that creates an array. The standard such loop is:

Fig. 15.1



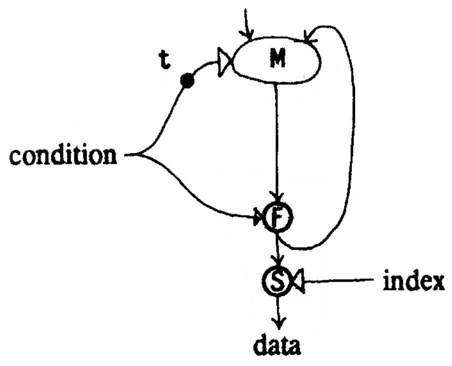
The "array packer" is of this type, as are many array-filling loops.

This loop can easily be seen to leave the reference count equal to one at all times, and yield as its result an array with reference count equal to one. Consequently, no run-time manipulation of reference counts needs to be performed, and an *in situ* array write can be performed for the **append**. If the initial array came in from some other computation with reference count equal to one, the situation would be the same.

If the array must pass through more than one **append**, as it must if the interlace is less than the product of the loop unfolding and the reference interval, the reference count is still preserved as the array passes through the **append**'s in sequence.

Now consider a loop that reads from an array with **select operators:**

Fig. 15.2

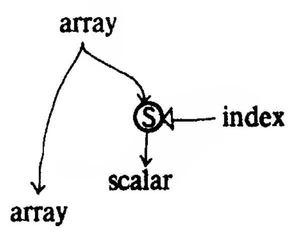


This obeys the reference count rule in spirit but not in fact. The token is duplicated just before the **select** operator. The **select** operator then reads from the array and destroys the array token, reducing the count back to one. A little thought will show that this is quite typical of array reading operations: When an array token goes into a **select**, it typically will be used again, so it must have been duplicated.

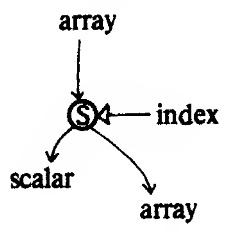
The excess reference count manipulations that are implied by this style of select can be avoided if the operator is redefined to have a second output arc. That arc emits the same array token that entered. Instead of duplicating the token and sending one copy to the **select** and keeping the other copy for later use, the correct procedure is to send the single token into the **select** and use the copy that it emits for later operations.

Fig. 15.3

Now instead of this



we have this



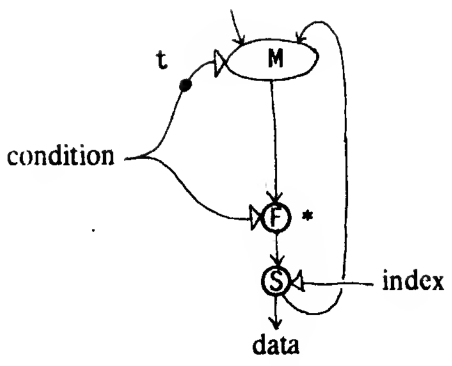
If the array must go to more than one **select**, as it must if the interlace is less than the product of the loop unfolding and the reference interval, it should go through these modified **select**'s in sequence, thereby preserving the reference count.

15.4 The "Create" and "Reclaim" Operators

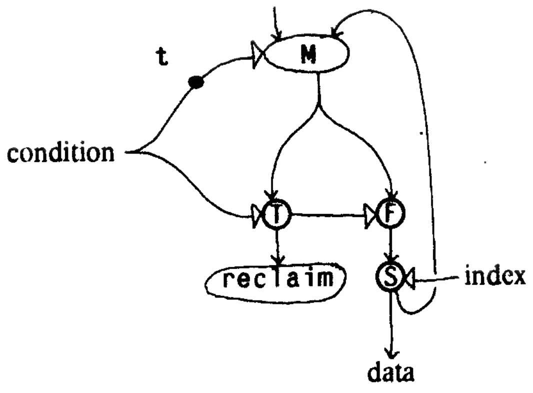
An array is destroyed, and its memory space reclaimed, when its reference count goes to zero. Assuming that the reference count is known to be one during the array's lifetime, this typically occurs at the end of a loop when the control value directs a gate to dispose of the token. Instead of having the gate take care of reclaiming the space, we can have a special operator do the job.

Fig. 15.4

instead of



we have



* This operator discards its token when the control value is true, so it must reclaim the array, which requires interaction with the memory system.

To create an array, some sort of **create** operator is appropriate, which could take a numeric argument telling how big a block of memory to allocate. Of course a well-rounded computer should be able to handle dynamic arrays and allow the allocation of an already existing array to increase beyond any predicted size, but in most cases the size can be predicted. Array creation with a predicted size is certainly a situation that the system should be able to handle efficiently. The prediction might be a constant, or might be the result of a previous computation.

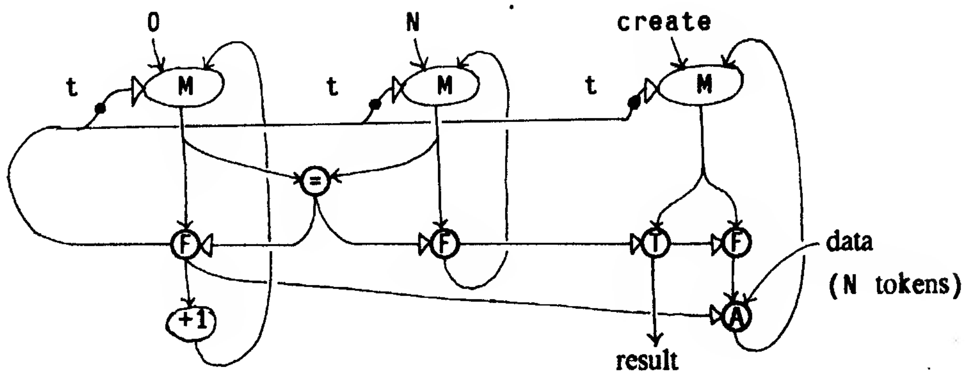
15.5 Fully Static Allocation

In many cases, an outer loop repeatedly causes an array to be created, used in an inner loop, and reclaimed. If the compiler can determine that the array is of the same size each time and that the next **create** operator does not need to act until after the previous **reclaim**, the **reclaim/create** pairs can be combined, and the same array token used repeatedly. Such transformations point the way toward fully static array allocation, a technique that is certainly efficient but not very general.

15.6 The "Pack" and "Unpack" Modules

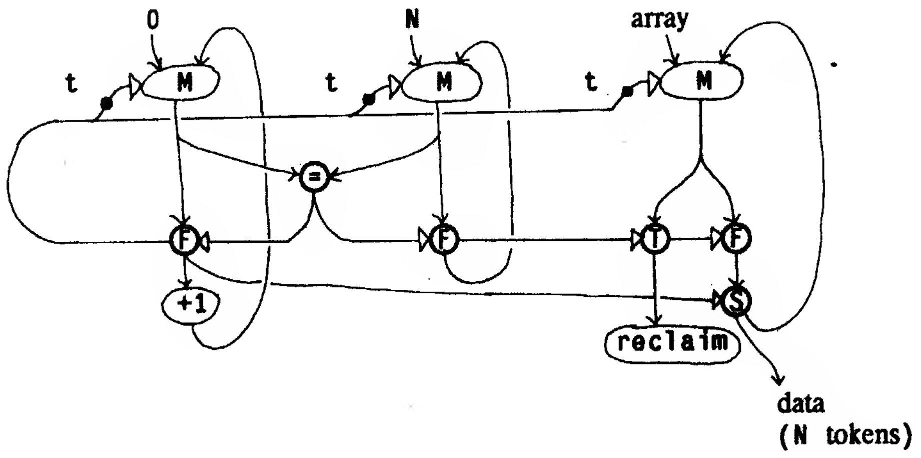
The array "packer" and "unpacker" are the prototypical modules for manipulating arrays. A "packer" for **N** items (where **N** is either constant or computed during execution) is, in full detail:

Fig. 15.5



An "unpacker" is

Fig. 15.6



The complexity of these subgraphs makes the point of this chapter: The "packer" and "unpacker" are probably the most common modules that manipulate arrays, but, if built out of the standard data flow operators, are far too unwieldy. A good machine design should support these operations with great efficiency in both time and space. Ideally, each of these subgraphs could be compressed into a single operator. Burkowski's instruction set [13] can compress the control parts of these graphs into efficient operators, but does not take the array operations themselves into consideration.

15.7 Boolean Pipelines

As discussed in Chapters 10 and 14, there are many places where it is useful to allow different parts of a loop to "decouple", letting the loop control section run several cycles ahead of some parts of the loop. This requires FIFO buffers in the lines between the control section and the rest of the loop body. Those lines go to "T", "F", and "MERGE" operators. It follows that a compiled program will have many such operators with FIFO buffers. Since the data in the buffers is of boolean type, the buffers really don't need much space. A good instruction set might arrange to have these operators, without taking up too much extra space, function as though they had built-in FIFO buffers of modest length.

16. CONCLUSION

We have seen how a number of program optimizations may be performed that, for many programs, should lead to extremely efficient execution on a suitably designed and sufficiently large computer. Many of the optimizations are "parameterized" -- they depend on choices of the degree of loop unfolding and array interlace. How might an optimizer make these choices, and how can we predict how well they will work?

An adequate theory of how well programs will perform under transformations of this type does not exist. The performance of a program on a parallel computer of the type we are considering may depend very strongly on seemingly minor points of data dependency. Programmers are not yet fully accustomed to thinking in terms of these points, so perhaps a detailed theory that would apply to a large class of programs is inappropriate at this time. The class of programs for which such a theory would be profitable has not been characterized in a useful way.

A great many programs have been analyzed, of course. However, the class of programs expressible in most languages is so rich that the problem seems intractable, at least at present. The best execution-time organization of a program, and the optimum values of the unfolding and lookahead, are somewhat "random" functions of the structure of the program.

We propose the following approach for an operational compiler: Very approximate values will be chosen for the parameters (unfolding and interlace) based on the need to match unfolding to interlace and on the constraint of total machine size. These values will then be refined, that is, the parameter space will be searched for nearby points that might yield better performance. The estimated performance at each point will be calculated from such things as the lengths of critical instruction chains and the number of cycles executed in various loops. The size of the unfolded program must also be estimated, to be sure it will fit in the computer. If these estimates can be made more accurate by using detailed knowledge about the computer's structure (relative speeds of floating point and fixed point instructions, behavior of the scheduler, behavior of interconnection networks, etc.) so much the better. Such detail is probably unnecessary, however. It could lead to the expenditure of more computation to optimize the program than to run it. Also, since the parameters are adjusted by factors of two, changes in performance should be quite noticeable and not require accurate measurements for their detection.

Using a very simple measure of efficiency, an optimizer should be able to find a point in the parameter space that will lead to reasonably nearly optimum performance. Exactly optimum program organization is unattainable without fully simulating the program. This is because events that depend on the data at execution time, such as arithmetic exceptions, can determine when certain parts of the program will be executed. Determination of the amount of the computer's resources to dedicate to these parts depends on the frequency with which these events will occur.

Given that the optimizer uses this "experimental" approach, the efficiency of an optimized program may depend, in perhaps unforeseen ways, on subtle details of the algorithm. A small change in the data dependencies specified in the program might not greatly affect the total efficiency, but it might greatly alter the parameters required to obtain the best efficiency. This is certainly a less than ideal situation, but it is not unlike the situation that presently exists in conventional systems. There is considerable "folklore" about the best ways to code various algorithms in various languages on various computer systems. A similar folklore might arise in the world of applicative numerical programming. There is some cause for optimism that this folklore will be more fundamental and less machine dependent than the corresponding knowledge about conventional systems. It is also likely that this knowledge will be able to be incorporated in a systematic way into future optimizing compilers.

REFERENCES

[1] Ackerman, W. B., *A Structure Memory for Data Flow Computers*, Laboratory for Computer Science (TR-186), MIT, Cambridge, Massachusetts, August 1977.

[2] Ackerman, W. B., "A Structure Controller for Data Flow Computers", Computation Structures Group (Memo 156), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, January 1978.

[3] Ackerman, W. B., "A Structure Processing Facility for Data Flow Computers", *Proceedings of the 1978 International Conference on Parallel Processing* (G. J. Lipovski, Ed.), August 1978, 166-172. Also, Computation Structures Group (Memo 165), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July 1978.

[4] Ackerman, W. B., "Data Flow Languages", *Computer 15*, 2(February 1982), 15-25. Also, Computation Structures Group (Memo 177-1), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, June 1981.

[5] Ackerman, W. B., and J. B. Dennis, *VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual*, Laboratory for Computer Science (TR-218), MIT, Cambridge, Massachusetts, June 1979.

[6] "Preliminary ADA Reference Manual", *SIGPLAN Notices 14*, 6(June 1979), .

[7] Allan, S. J., and A. E. Oldehoeft, "A Flow Analysis Procedure for the Translation of High Level Languages to a Data Flow Language", *Proceedings of the 1979 International Conference on Parallel Processing* (O. N. Garcia, Ed.), August 1979, 26-34.

[8] Allen, F. E., and J. Cocke, "A Catalogue of Optimizing Transformations", in *Design and Optimization of Compilers*, (R. Rustin, Ed.), Prentice-Hall, 1972.

[9] Arvind, K. P. Gostelow, and W. Plouffe, *An Asynchronous Programming Language and Computing Machine*, Department of Information and Computer Science (TR 114a), University of California - Irvine, Irvine, California, December 1978.

[10] Arvind, and R. E. Thomas, "I-Structures: An Efficient Data Type for Functional Languages", Laboratory for Computer Science (TM-178), MIT, Cambridge, Massachusetts, September 1980.

[11] Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM 21*, 8(August 1978), 613-641.

[12] Barnes, G. H. et al, "The Illiac IV Computer", *IEEE Transactions on Computers C-17*, 8(August 1968), 746-757.

[13] Burkowski, F. J., "Instruction Set Design Issues Relating to a Static Dataflow Computer", *Proceedings of the Ninth Annual Symposium on Computer Architecture, SIGARCH Newsletter 10*, 3(April 1982), 101-111.

[14] Aho, A. V., and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1979.

[15] Cooley, J. W., and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series", *Mathematics of Computation 19*, 90(April 1965), 297-301.

[16] Cornish, M., D. W. Hogan, and J. C. Jensen, "The Texas Instruments Distributed Data Processor", *Proceedings of the Louisiana Computer Exposition*, March 1979, 189-193.

[17] Davis, A. L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine", *The Fifth Annual Symposium on Computer Architecture, SIGARCH Newsletter 6*, 7(April 1978), 210-215.

[18] Dennis, J. B., "First Version of a Data Flow Procedure Language", *Lecture Notes in Computer Science, 19*, Springer-Verlag, New York, 1974, 362-376. Also, Laboratory for Computer Science (TM-61), MIT, Cambridge, Massachusetts, May 1975.

[19] Dennis, J. B., "Packet Communication Architecture", *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, August 1975, 224-229. Also, Computation Structures Group (Memo 130), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1975.

[20] Dennis, J. B., "The Varieties of Data Flow Computers", *The First International Conference on Distributed Computing Systems*, October 1979, 430-439. Also, Computation Structures Group (Memo 183), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1979.

[21] Dennis, J. B., "Data Flow Supercomputers", *Computer 13*, 11(November 1980), 48-56.

[22] Dennis, J. B., G. A. Boughton, and C. K. C. Leung, "Building Blocks for Data Flow Prototypes", *The Seventh Annual Symposium on Computer Architecture*, May 1980, 1-8.

[23] Dennis, J. B., and J. B. Fosseen, "Introduction to Data Flow Schemas", Computation Structures Group (Memo 81), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, September 1973.

[24] Dennis, J. B., J. B. Fosseen, and J. P. Linderman, "Data Flow Schemas", *International Symposium on Theoretical Programming* (A. Ershov, V. A. Nepomniashy, Eds.), *Lecture Notes in Computer Science 5*, August 1972, 187-216.

[25] Dennis, Jack B., Guang-Rong Gao, and Kenneth W. Todd, "A Data Flow Supercomputer", Computation Structures Group (Memo 213), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, February 1982.

[26] Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor", *The Second Annual Symposium on Computer Architecture: Conference Proceedings*, January 1975, 126-132. Also, Computation Structures Group (Memo 102), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1974.

[27] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, 1976.

[28] Flynn, M. J., "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers C-21*, 9(September 1972), 948-960.

[29] American National Standards Institute, "American National Standard Programming Language FORTRAN X3.9-1978",

[30] Friedman, D. P., and D. S. Wise, "The Impact of Applicative Programming on Multiprocessing", *Proceedings of the 1976 International Conference on Parallel Processing* (P. H. Enslow, Ed.), August 1976, 263-272.

[31] Gajski, D. D., et al, "A Second Opinion on Data Flow Machines and Languages", *Computer 15*, 2(February 1982), 58-69.

[32] Gao, G-R., "An Implementation Scheme for Array Operations in Static Data Flow Computers", Laboratory for Computer Science (TR-280), MIT, Cambridge, Massachusetts, May 1982.

[33] Gurd, J., I. Watson, and J. Glauert, "A Multilayered Data Flow Computer Architecture", Department of Computer Science, University of Manchester, Manchester, England, July 1978.

[34] Henderson, P., and J. Morris, Jr., "A Lazy Evaluator", *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, January 1976, 95-103.

[35] Hoare, C. A. R., "Communicating Sequential Processes", *Communications of the ACM 21*, 8(August 1978), 666-677.

[36] Hockney, R. W., "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis", *Journal of the ACM 12*, 1(January 1965), 95-113.

[37] Kahn, G., "The Semantics of a Simple Language for Parallel Programming", *Information Processing 74: Proceedings of IFIP Congress 74* (J. L. Rosenfeld, Ed.), August 1974, 471-475.

[38] Kuck, D. J., R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimization", *Proceedings of the Eighth Symposium on Principles of Programming Languages*, January 1981, 207-218.

[39] Kung, H. T., and C. E. Leiserson, *Systolic Arrays for VLSI*, Department of Computer Science, Carnegie-Mellon University (CMU-CS-79-103), April 1978.

[40] Keller, R. M., G. Lindstrom, and S. S. Patil, "A Loosely-Coupled Applicative Multi-processing System", *Proceedings of the 1979 National Computer Conference, AFIPS Conference Proceedings 48*, June 1979, 613-622.

[41] Lawrie, D. H., "Access and Alignment of Data in an Array Processor", *IEEE Transactions on Computers C-24*, 12(December 1975), 1145-1155.

[42] McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine", *Communications of the ACM 3*, 4(April 1960), 185-195.

[43] Siegel, H, J., "Interconnection Networks for SIMD Machines", *Computer 12*, 6(June 1979), 57-65.

[44] Stone, H. S., "Parallel Tridiagonal Equation Solvers", *ACM Transactions on Mathematical Software 1*, 4(December 1975), 289-307.

[45] Turner, D. A., "The Semantic Elegance of Applicative Languages", *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, October 1981, 85-92.

[46] Weng, K.-S., "Stream-Oriented Computation in Recursive Data Flow Schemas", Laboratory for Computer Science (TM-68), MIT, Cambridge, Massachusetts, October 1975.